Towards Introducing Asynchronous Tasks to an FRP Language for Small-Scale Embedded Systems

Akihiko Yokoyama
akihiko@psg.c.titech.ac.jp
Tokyo Institute of Technology
Tokyo, Japan

Sosuke Moriguchi
chiguri@acm.org
Tokyo Institute of Technology
Tokyo, Japan

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

Abstract

Emfrp is a functional reactive programming language designed for small embedded systems. By imposing certain restrictions on the language mechanism, the language guarantees the termination of the update process for each time-varying value and enables static determination of the amount of memory required for execution. This allows Emfrp programs to run safely even in resource-constrained execution environments. However, the abovementioned restrictions make it difficult to write time-consuming operations (heavy tasks) such as graph structure construction and exploration based on external data. Moreover, since Emfrp updates time-varying values synchronously, a naive implementation of such heavy tasks using external function calls will result in a slow response time to input. Some existing programming languages provide asynchronous processing mechanisms to ensure descriptiveness and responsiveness for heavy tasks. In this study, we propose a method to introduce heavy tasks into reactive programs naturally by introducing language mechanisms equivalent to asynchronous processing mechanisms, such as future and promise, into Emfrp. In this paper, we first discuss the problems with a naive implementation of heavy tasks in Emfrp, then explain the proposed method based on an example, and discuss the language runtime implementation.

CCS Concepts: • **Computer systems organization** → *Embedded software*; • **Software and its engineering** → **Functional languages**.

Keywords: functional reactive programming, asynchronous tasks, promise, embedded systems

ACM Reference Format:
Akihiko Yokoyama, Sosuke Moriguchi, and Takuo Watanabe. 2022. Towards Introducing Asynchronous Tasks to an FRP Language for Small-Scale Embedded Systems. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS '22), December 07, 2022, Auckland, New Zealand.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3563837.3568338

REBLS '22, December 07, 2022, Auckland, New Zealand

1 Introduction

A *reactive system* is a computing system that continuously responds to external inputs given synchronously or asynchronously while changing its internal state. In implementing reactive systems using conventional programming languages, techniques such as polling and callbacks (interrupt handling) are commonly used. However, these techniques often reduce program readability and maintainability [3]. *Functional Reactive Programming* (*FRP*) is a programming paradigm that supports clear and declarative descriptions of reactive systems by using *time-varying values* (*signals*) that abstract values that change over time. FRP was originally proposed as an interactive animation library for Haskell [8] and has since been studied and shown to be beneficial in various fields such as GUIs [6], web applications [7], robotics [13, 19], embedded systems [12, 24], and IoT [23].

Emfrp [24] is an FRP language designed for small-scale embedded systems. The memory footprint of the executable code of the language is small enough to be executable in resource-constrained execution environments such as microcontrollers. Also, to run programs safely in such environments, the language is designed to have certain restrictions (see Section 3.1) on the syntax and type system to guarantee the termination of the update process for each time-varying value and to allow static determination of the amount of runtime memory. We have so far proposed various extensions to Emfrp to explore the advantages of the language in various applications [20, 21, 27, 28].

In FRP, it is often assumed that the time required to update each time-varying value is negligible. However, even in small-scale embedded systems, there are cases where this assumption does not hold. Time-consuming operations (called *heavy tasks* throughout this paper), such as the graph construction and search described in Section 3, in the updating process of time-varying values in a system have a negative

impact on the reactivity of the system by increasing its response time. Although the language restrictions mentioned above limit the ability to write heavy tasks directly in Emfrp, they can be introduced through the naïve use of external (foreign) functions.

The Actor-Reactor model [26] provides a solution to the problem. The model isolates operations corresponding to heavy tasks by introducing a different execution model (the Actor model [1]), thereby eliminating the negative impact on reactive behavior. However, this approach may reduce the readability of the program because it splits interdependent reactive behaviors and heavy tasks.

In this paper, we introduce an extension of Emfrp that includes a mechanism for describing asynchronous tasks and a type corresponding to futures or promises. The main contributions of this work are (1) to show, through a non-trivial example, that the proposed extension allows heavy tasks to be embedded in a reactive code naturally and (2) to present the implementation method of the runtime system for the extended language on resource-constrained embedded systems.

The rest of the paper is organized as follows. Section 2 provides an overview of Emfrp, followed by an example that motivates this study in Section 3. The problems of naïve solutions are also clarified in this section. Section 4 describes the proposed extension of Emfrp and its runtime system implementation. The section also provides a description of the example using the extension. Section 5 discusses the relationship with related work, and Section 6 provides the summary and future work.

2 Emfrp

This section presents an overview of Emfrp [24], an FRP language designed for small-scale embedded systems. For detailed descriptions, please refer to the paper [24] and the source code repository [1].

2.1 Example: A Differential Drive Robot

Figure 1 is an Emfrp version of the two-wheeled differential drive robot example in Yampa [13]. This program continuously computes the position of the robot. Let $v_l(t)$ and $v_r(t)$ be the velocities of the left and right wheels at time t, respectively. The position $(x(t), y(t))$ and direction $\theta(t)$ of the robot are:

$$x(t) = \frac{1}{2}\int_0^t (v_l(u) + v_r(u)) \cos\theta(u) du \quad (1)$$

$$y(t) = \frac{1}{2}\int_0^t (v_l(u) + v_r(u)) \sin\theta(u) du \quad (2)$$

$$\theta(t) = \frac{1}{l}\int_0^t (v_r(u) - v_l(u)) du \quad (3)$$

where l is the distance between the left and right wheels.

[1] https://github.com/psg-titech/emfrp

```
1  # RobotPos.mfrp
2  module RobotPos    # Module Name
3  in  vl : Float,    # Left velocity [m/sec]
4      vr : Float,    # Right velocity [m/sec]
5      t(0) : Int     # Elapsed time [msec]
6  out x : Float,     # X-coordinates [m]
7      y : Float      # Y-coordinates [m]
8  use Std, Params    # Import library
9
10 node dt = (t - t@last) / 1000.0
11 node init [0.0] theta =
12   theta@last + (vr - vl) * dt / l
13
14 node init[0.0] x =
15   x@last + (vr + vl) * cos(theta) * dt / 2.0
16 newnode y =
17   CalcPosY(vl, vr, theta, dt) # using submodule
```

```
1  # Params.mfrp
2  material Params
3
4  # Constant value
5  data l = 0.1 # Wheel-to-wheel distance [m]
6
7  # function
8  func max(a: Int, b: Int) = if a > b then a else b
```

```
1  # CalcPosY.mfrp
2  module CalcPosY
3  in  vl: Float, vr: Float, theta: Float, dt: Float
4  out y : Float
5  use Std
6
7  node init[0.0] y =
8    y@last + (vr + vl) * sin(theta) * dt / 2.0
```

Figure 1. Position of a differential drive robot

The program in Figure 1 consists of two modules (in files `RobotPos.mfrp` and `CalcPosY.mfrp`) and a library (in file `Params.mfrp`). A module consists of a header and a body. The header section (lines 2–8 in `RobotPos.mfrp`, lines 2–5 in `CalcPosY.mfrp`) declares the module name, input and output nodes, and the name of required libraries. The body section (lines 10–17 in `RobotPos.mfrp`, line 7–8 in `CalcPosY.mfrp`) consists of the definitions of constants, functions, intermediate nodes and output nodes, and the instantiations of submodules.

2.2 Time-Varying Values (Nodes)

Time-varying values in FRP are represented by objects called *nodes* in Emfrp. Nodes are categorized as *input nodes*, *output nodes*, and *intermediate nodes*. In the `RobotPos` module, `vl`, `vr` and `t` are input nodes, `x` and `y` are output nodes, and `dt` and `theta` are intermediate nodes.

Input nodes receive values from external sources. For example, `vl` and `vr` receive the values from the rotary encoders connected to the left and right wheels respectively, and `t` receives the value from the system timer.

Intermediate nodes and output nodes have update expressions defined by the syntax **`node`** $n = e$ or **`node init`**`[`c`]` $n = e$,

```c
// RobotPosMain.c
void Input(float* vl, float* vr, int* t){
  /* Get values from sensors */
}

void Output(float x, float y){
  /* Put values to actuators */
}

int main(void){
  // Activate Emfrp's RobotPos module
  // This function is generated from RobotPos
    module by compiler.
  ActivateRobotPos();
  return 0;
}
```

Figure 2. Template I/O code for `RobotPos` module

where n is the node's name, e is the expression, and c is a constant.

One of the features of Emfrp is that the *previous value* of a node can be obtained by using the n`@last` expression. Using the previous values makes it easy to calculate the difference and cumulative time-varying values. For example, `theta`, `x` and `y` use `@last` to approximate the integral as the cumulative values of a small amount of data. Nodes to which the previous value may be referred are set to their initial values. For intermediate and output nodes, node definitions with **`init`** are used. For input nodes, initial values are set in the module's header, as in `RobotPos.mfrp` line 5.

If a definition of node x contains a reference to node y (y`@last` is not considered a reference to node y), we say that node x depend on node y. In Emfrp, all nodes in the program are statically checked to ensure that their dependencies are acyclic directed graphs (DAGs) to determine the correct order of time-varying value updates.

2.3 Input/Output with External Environment

The Emfrp program is compiled into C source code. At the same time, a template code that contains functions for input/output and a function call to invoke the Emfrp program is emitted. Figure 2 is an excerpt of the template code output when the `RobotPos` module is compiled as a top-level module. The Emfrp program starts by calling the `ActivateRobotPos` function in the `main` function. `Input` function is called when external input is required during the processing of the function `ActivateRobotPos`. Similarly, `Output` function is called when external output is required.

2.4 Execution Model

Emfrp performs a push-based time-varying update process. This execution model propagates input node changes to dependent nodes in order. Since it is checked at compile time that there are no loops in the node dependencies, the time-varying value update process can be performed appropriately through the order of the dependencies. In Emfrp, the sequence of processing is as follows: input function call, input node update, internal and output node update in the order of dependencies, output function call, the previous value update, and memory management. We call this sequence iteration. Repeating this iteration process sequentially without a break makes it possible to perform reactive behavior.

3 Motivation

In this section, we explain the need for language extensions in Emfrp, using an example that requires coordination between reactive behavior and heavy task (see below) execution.

3.1 The Restrictions of Emfrp

Emfrp has some language restrictions. They prevent time leaks and space leaks. Moreover, it also guarantees that the program will not cause runtime errors due to insufficient memory and that the program keeps reactive behavior. Some restrictions are: restricting past value references of time-varying values to the previous value, prohibiting higher-order time-varying values (that are time-varying values of time-varying values), and disallowing recursive definitions of functions and data types. Arrays are also not introduced due to concerns about out-of-range access errors. Due to these restrictions, the node update expressions are relatively simple code. Therefore, each node can be updated instantaneously, and the continuous execution of iterations ensures sufficient responsiveness.

3.2 Heavy tasks

Even small-scale embedded systems often perform tasks that require the use of complex data structures such as graphs. Using such data structures can easily lead to memory bloat and increased computation time. If node update computations become extremely slow, the responsiveness of the whole system will deteriorate. The problem that slow computations deteriorate system responsiveness is called the *Reactive Thread Hijacking Problem (RTHP)* [26]. In this paper, we call the computations not required to be executed every iteration but cause RTHPs"heavy tasks." It is assumed that writing to and reading from data structures such as graphs is performed in heavy task.

The restrictions of Emfrp described in the previous section make it challenging to define and manipulate complex data structures such as graphs. Therefore, it is tough to implement heavy tasks in Emfrp. *EmfrpBCT* [28], which introduces a bounded-size recursive data type to Emfrp, allows the construction and manipulation of lists and tree structures with a fixed maximum size, thus allowing the writing of heavy tasks. However, in-place updating of data structures is not supported, so the update process duplicates the data

structure multiple times. In a resource-constrained environment, the RTHP and the problem of reducing the size of the available data structure are simultaneously caused. For this reason, heavy tasks on Emfrp (BCT) are often written using FFI to C. Since naive FFI cannot solve RTHP, however, an asynchronous task execution mechanism is needed to coordinate reactive behavior, which is responsive enough, with heavy task execution, which takes time.

3.3 Motivating Example: Exploration Robot

Let us take a maze-exploring robot as an example of a system requiring reactive actions and heavy task execution coordination. This example mimics the maze traversal of the Micromouse robot competition [2]. The robot with sensors and drive wheels explores a given unknown maze and aims to move autonomously from the start section to the goal section.

3.3.1 Problem Setting. The maze is rectangular and divided into square sections by walls. Figure 3 shows an example of a maze. The walls surround the entire perimeter of the maze. The robot is given the size of the maze and its sections, the number of sections, the position of the start and goal section, and the layout of the walls of the start section in advance. The start position is fixed at the lower left corner of the maze, and the walls of the start section are located in the east-south-west direction, with no walls on the north side. The robot does not know the information about the walls in the maze.

Figure 4 shows an illustration of the robot from the top. The robot has four infrared sensors to detect walls, three motors with omni-wheels [3], and one LED to indicate that the robot has reached its goal. Each motor is equipped with a rotary encoder that can measure rpm. The omni-wheel consists of a standard wheel with multiple wheels on its circumference and can move in a direction perpendicular to the direction of rotation of the wheels. The robot can move its body in any direction by arranging the omni-wheels as shown in Figure 4 and applying the appropriate motor outputs to each. In other words, the robot in this example is an omni-directional mobile vehicle and does not rotate its body when moving between sections.

We explain how the robot moves from the start section to the goal section. First, the robot is placed in the start section and begins its program. Since the wall layout of the start section is fixed, the robot moves to the section to the north of the start section, which is the target section. When the center of the robot body moves to the center of the target section, the robot recognizes that it has reached the target section. Then the robot records the wall information of the target section. After reaching the target section, the robot executes a maze exploration algorithm with the wall information recorded until now. Note that the next target section is only one section away from the current section in either the east, west, south, or north direction. The A* algorithm or the extended left-hand method can be used as a maze exploration algorithm. The robot moves to the new target section obtained from the algorithm. The robot eventually arrives at the goal section by repeating these actions. When the robot reaches the goal section, it turns on an LED to notify the user that it has finished its execution.

These actions consist of (1) moving the robot to the target section, (2) recording the wall information, and (3) calculating the next destination by the exploration algorithm with the recorded wall information. Action (1) is a reactive action with the motor speed, target coordinates, and elapsed time as inputs, the current position as the internal state, and the motor power level as the output. In contrast, actions (2) and (3) are heavy task actions because they modify the graph structure representing the maze and perform an exploration algorithm with the graph.

The robot described above invokes actions (1), (2), and (3), in sequential order. However, as shown in Figure 5, if the center of the robot's body is somewhat close to the center of the target section, the wall information of the target section can be obtained. Thus, the robot can record the wall information in parallel with its movement toward the target section and then execute the maze exploration algorithm to obtain the next target section. Suppose the recording of wall information and the exploration algorithm is completed before the robot's center moves to the center of the target section. In that case, the robot can instantly move on to the next target section after reaching the target section, thus reducing the time the robot spends traversing the maze. The robot's exploration performance can be improved by coordinating reactive behavior and heavy task execution. In the following, the exploration in which reactive behaviors and heavy task execution are repeated sequentially is called the standard exploration. The one in which reactive behaviors and heavy task execution are executed concurrently is called the improved exploration.

3.3.2 Writing the Example in Plain Emfrp. We write the robot control program described above in Emfrp (without extensions). Figure 6 shows the Emfrp module `MovePID` that moves the center of the robot's body to the target position by PID control. This module takes as input the target position (`tar_x`, `tar_y`), the rotation count of each motor (`enc1`, `enc2` , `enc3`), and the elapsed time since the previous iteration (`duration`). The output is the distance to the target position (`distance`) and the power ratio of each motor (`duty1`, `duty2`, `duty3`). The module internally holds the current position of the robot (`cur_x`, `cur_y`) as a node and continues to update it using sensor values. It calculates the control variables (output ratio of motors) to approach the target position by PID control. More precisely, the robot's rotation also needs

[2] https://en.wikipedia.org/wiki/Micromouse
[3] https://en.wikipedia.org/wiki/Omni_wheel

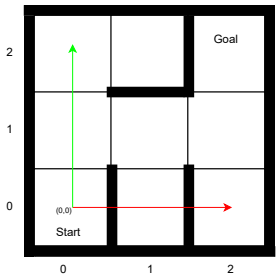

Figure 3. Example of a maze.

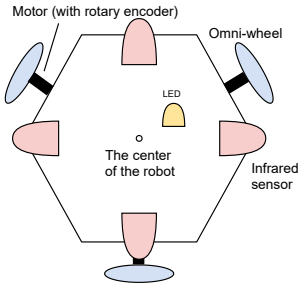

Figure 4. The exploration robot.

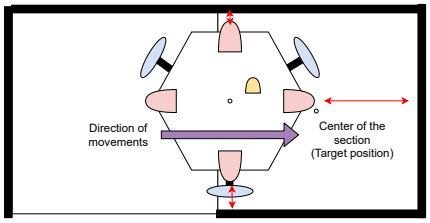

Figure 5. The robot moves from the left to the target (right) section.

```
1  module MovePID # Position controller
2  in tar_x : Float, tar_y : Float, # Target
3     enc1: Int, enc2: Int, enc3: Int,
4     duration : Float # Elapsed time [sec]
5  out distance : Float,
6      duty1 : Int, duty2 : Int, duty3 : Int
7  use Std
8
9  # Coefficients of PID
10 data (Kp, Ki, Kd) = ...
11 # From motor rpm to movement vector.
12 func motor_to_vector(...) = ...
13 # From movement vector to power ratio.
14 func motor_speed(...) = ...
15
16 # updating current positions.
17 node (mx, my) = motor_to_vector(enc1, enc2, enc3)
18 node init [(0, 0)] (cur_x, cur_y) =
19   (cur_x@last + mx, cur_y@last + my)
20
21 # PID control for X axis.
22 node dx = Kp * e_x + Ki * ei_x + Kd * ed_x
23 node init [0] e_x = tar_x - cur_x
24 node init [0] ei_x =
25   ei_x@last + (e_x + e_x@last) / 2 * duration
26 node ed_x = if duration == 0 then 0
27   else (e_x - e_x@last) / duration
28
29 # PID control for Y axis (same as X).
30 node dy = Kp * e_y + Ki * ei_y + Kd * ed_y
31 node (e_y, ei_y, ed_y) = ...
32
33 # Distance to target
34 node distance = sqrt(e_x*e_x + e_y*e_y)
35 # Output to motors
36 node (duty1, duty2, duty3) = motor_speed(dx, dy)
```

Figure 6. Position Controller by PID control

```
1  module MazeRunnerEmfrp
2  in sn : Int, se : Int, ss : Int, sw : Int,
3     enc1: Int, enc2: Int, enc3: Int,
4     time(0) : Float, # Time from start [sec].
5     to_u : Int, to_v : Int
6  out finished : Bool,
7      duty1 : Int, duty2 : Int, duty3 : Int,
8      reached_target : Bool,
9      wall_n : Bool, wall_e : Bool,
10     wall_s : Bool, wall_w : Bool
11 use Std
12
13 data G_U = ... # X-index of goal section.
14 data G_V = ... # Y-index of goal section.
15 data WS = ... # The width of sections.
16 data THR_WALL = ... # Threshold of sensors.
17 data THR_TARGET = ... # Epsilon distance.
18
19 # Time taken for the previous iteration.
20 node duration = time - time@last
21
22 # Control position.
23 newnode target_dist, d1, d2, d3 =
24   MovePID(to_u * WS, to_v * WS,
25           enc1, enc2, enc3, duration)
26
27 node duty1 = if reached_target then 0 else d1
28 node duty2 = if reached_target then 0 else d2
29 node duty3 = if reached_target then 0 else d3
30
31 # Whether the robot reached target section.
32 node reached_target = target_dist < THR_TARGET
33
34 # Wall information
35 node wall_n = sn > THR_WALL
36 node wall_e = se > THR_WALL
37 node wall_s = ss > THR_WALL
38 node wall_w = sw > THR_WALL
39
40 # Whether the robot reached goal.
41 node finished = (to_u == G_U && to_v == G_V
42   && reached_target)
```

Figure 7. The maze exploration robot in Emfrp.

to be controlled, but here we assume that the robot does not rotate. If this is the case, PID control of the robot's rotation can be performed using the rotary encoder and gyro sensor values.

Figure 7 shows the Emfrp module `MazeRunnerEmfrp`, which performs the standard exploration. Figure 8 shows the pseudo code of the input and output functions in C for the module `MazeRunnerEmfrp`. The `MazeRunnerEmfrp` module takes as input the values of the infrared sensors in the four directions (`sn`, `se`, `ss`, `sw`), the rotation counts of the motors (`enc1`, `enc2`, `enc3`), the time since the start of the program (`time`), and the index of the target section (`to_u`, `to_v`). In line 23, the motor outputs (`d1`, `d2`, `d3`) are calculated to move the robot's body to the center of the input target section (`to_u`, `to_v`) by using `MovePID` module as a submodule. When the robot reaches the target section, each motor output is set to 0 to stop the robot

```
1  // Global variables across Input and Output
2  int next_u = 0; // X-index of the next target
3  int next_v = 1; // Y-index of the next target
4  bool has_next = false;
5
6  void Input(int* sn, int* se, int* ss, int* sw,
7    int* enc1, int* enc2, int* enc3,
8    float* time, int* to_u, int* to_v){
9    get_sensors(sn, se, ss, sw);
10   get_encoders(enc1, enc2, enc3);
11   get_time(time);
12   if(has_next){
13     *to_u = next_u; *to_v = next_v;
14     has_next = false;
15   }
16 }
17
18 void Output(bool finished,
19   int duty1, int duty2, int duty3,
20   bool reached_target, bool wall_n,
21   bool wall_e, bool wall_s, bool wall_w){
22   if(finished) set_led();
23   set_motor(duty1, duty2, duty3);
24   if(reached_target){
25     // Heavy task executions
26     set_wall(next_u, next_v,
27              wall_n, wall_e, wall_s, wall_w);
28     calc_next(&next_u, &next_v);
29     has_next = true;
30   }
31 }
```

Figure 8. Input and Output functions for `MazeRunnerEmfrp`

at 27–29 lines. Next, in Figure 8, the next target section calculated by the `Output` function is fed back to the `Input` function via the global variables (`next_u`, `next_v`, `has_next`). If the output node `reached_target` is `true`, the robot has reached the target section and has stopped. This condition is detected at line 24 (in the `Output` function) of Figure 8. Then, the wall information at that time is registered (`set_wall` function), and the next target section is calculated (`calc_next` function). During the calculation, the Emfrp iteration process is suspended. When the calculation is finished, the `Output` function is done, and the Emfrp iteration starts again. At the next iteration, when the `Input` function is invoked, the global variable `has_next` is `true`, so the new target section is set. Since the current position and the new target section are far apart, the `reached_target` node will be false, and the iteration will be executed to approach the target section.

We successfully implemented standard exploration by running heavy task in the output function and feeding back the result to the input function. On the other hand, it is challenging to implement the improved exploration due to executing heavy task sequentially in the output function. The problem (P1): the sequential execution of heavy tasks in the output function makes other reactive behaviors less responsive. Moreover, because the heavy tasks in C cause feedback from the output function to the input function, an implicit dependency from the output node to the input node inevitably appears. The problem (P2): such implicit dependencies and the data structures subject to heavy task are not explicitly indicated in the Emfrp program.

3.3.3 Coordination between Emfrp and RTOS.

To solve the problem (P1), let us use a library that supports asynchronous task execution, such as RTOS, and implement an improved exploration by executing heavy tasks concurrently with Emfrp iterations. In this case, the heavy task execution is interrupted during the iteration, so the iteration interval is expected to be longer. The iteration is not expected to be suspended for long, which solves the responsiveness problem (P1). However, the problem (P2) has not been solved because the heavy task execution part is written in C. In addition, data must be shared between the Emfrp iteration and the RTOS task, which is expected to use conventional concurrent programming techniques. Problem (P3): the difficulties of conventional concurrent programming become apparent, and the advantages of using Emfrp (or FRP) are lost.

4 Language and Runtime Extensions

To overcome the problems (P1), (P2), and (P3) while combining reactive behavior and heavy task execution, we introduce an asynchronous task execution mechanism to Emfrp. This section explains the language and runtime extensions through an example of an exploration robot written in Emfrp with the asynchronous task execution mechanism.

4.1 Language Extensions

Figure 9 shows an example of an exploration robot written in the extended Emfrp language. Figure 10 shows the materials imported into this module. Hereafter, asynchronous tasks representing heavy tasks are referred to as tasks, and the data structures targeted by the tasks are referred to as task resources.

The additional language features are: (1) syntax for defining tasks and task resources (Figure 10, lines 14–25), (2) mechanism for passing task resources between modules (Figure 9, line 3), (3) `Future` type that represents the computation state of a task, and (4) syntax for binding a task to a `Future` node (Figure 9, lines 15–17 and 20–21).

4.1.1 Tasks and Task Resources.

Tasks and task resources are defined in Figure 10, lines 14–25. The first part of the definition is the name of the task resource. A task resource is an abstraction of data handled by a task, and an instance of the task resource is manipulated at runtime. The relationship between a task resource definition and a task resource instance is similar to that between a class and its instance in Java (or other OOP language).

A task is defined in the form:

$$\texttt{task}\ g : (x_1 : \bar{\tau}, \ldots, x_n : \bar{\tau}) \texttt{->} (y_1 : \bar{\tau}, \ldots, y_m : \bar{\tau}) / p$$

where g is the task name, x_i is the name of the input, y_i is the name of the output, $\bar{\tau}$ is a type other than `Future`

```
1  # MazeRunner.mfrp
2  module MazeRunner
3  in mg : resource MazeGraph,
4     sn : Int, se : Int, ss : Int, sw : Int, enc1: Int, enc2: Int, enc3: Int, time(0) : Float
5  out finished : Bool, duty1 : Int, duty2 : Int, duty3 : Int
6  use Std, Resources
7
8  node duration = time - time@last
9  newnode target_dist, duty1, duty2, duty3 = MovePID(to_u * WS, to_v * WS, enc1, enc2, enc3, duration)
10
11 # Whether the robot has neared the target section
12 node init[False] near_target = target_dist < SECTION_WIDTH * 0.4
13 node reached_target = target_dist < THR_TARGET # Whether the robot reached target section.
14
15 tasknode finish_register : Future[Unit] =  # Register section information
16   RegisterSection(to_u, to_v, wall_n, wall_e, wall_s, wall_w) with mg
17     at (!near_target@last && near_target)  # positive edge of near_target
18
19 # Search result (if it has already reached the goal, keeps returning the goal coordinates)
20 tasknode next : Future[(Int, Int)] =
21   CalcNextSection(to_u, to_v, G_U, G_V) with mg at wait(finish_register)
22
23 # Which direction the robot is going
24 node move_dir = (to_u - from_u, to_v - from_v) of:
25   (0, 0) -> Stop,
26   (0, 1) -> StoN, (0, -1) -> NtoS, (1, 0) -> WtoE, (-1, 0) -> EtoW,
27   _ -> Stop # unreachable
28
29 node wall_n = move_dir of:  # Whether there is a wall to the north in target section
30   Stop -> sn > THR_WALL_SHORT,
31   StoN -> sn > THR_WALL_LONG,
32   NtoS -> False, # Absolutely no walls.
33   WtoE -> sn > THR_WALL_SHORT,
34   EtoW -> sn > THR_WALL_SHORT
35 node wall_e = ... # omitted
36 node wall_s = ...
37 node wall_w = ...
38
39 # Update target section (always move forward to (0, 1) at first)
40 # Define nodes of:
41 #  previous target (from_u, from_v), current target (to_u, to_v), temporarily next target (tmp)
42 node init[(0, 0, 0, 1, None)] (from_u, from_v, to_u, to_v, tmp) = (reached_target, next, tmp@last) of:
43     (True,  Ready((nu, nv)), _) -> (to_u@last, to_v@last, nu, nv, None),
44     (False, Ready(n),        _) -> (from_u@last, from_v@last, to_u@last, to_v@last, Some(n)),
45     (True,  _,  Some((nu, nv))) -> (to_u@last, to_v@last, nu, nv, None),
46     _ -> (from_u@last, from_v@last, to_u@last, to_v@last, tmp@last)
47
48 node finished = move_dir of:  # Whether the robot reached goal section
49   Stop -> to_u == G_U && to_v == G_V && reached_target,
50   _ -> False
```

Figure 9. The maze exploration robot in extended Emfrp.

type (see below), and p is either **read** or **write**. p is a hint to the compiler and runtime to avoid data races when multiple tasks concurrently handle the same task resource instance. In this example, for an instance of the task resource `MazeGraph`, the writing task is `RegisterSection` and the reading task is `CalcNextSection`.

In Figure 9, line 3, an instance of a task resource is passed as an argument to the module. The received instance can be passed to a submodule, which can define **tasknode** (see below) for that instance. In the case of a top-level module such as the `MazeRunner` module, the resource instance is passed at program activation (as described in the 4.2 section).

4.1.2 Future Type and Tasknode.

Nodes representing the result of a task need to indicate that the computation is unfinished since the task is executed asynchronously. In concurrent programming, it is typical to introduce a data type called "future" and type it to expressions that have not yet finished computation. We introduced `Future` type defined as:

```
type Future[A] = Ready(A) | Pending | NotStarted
```

where `A` is the type of task result. Nodes with type `Future` will only have the value of `Ready(_)` during the iteration immediately after its task has been completed. `Pending` indicates that the task has been executed (or waiting to be executed)

```
# Resources.mfrp
material Resources;
type MoveDir = Stop | StoN | NtoS | WtoE | EtoW
data THR_WALL_SHORT = ... # Threshold for wall
data THR_WALL_LONG = ...  # Threshold for wall
data (G_U, G_V, WS, THR_TARGET) = ...

# Function to wait for task completion.
func wait(s : Future[A]) = s of:
  Ready(_) -> True,
  Pending -> False, NotStarted -> False

# Define task resources and the tasks.
resource MazeGraph {
  # Task to record wall information of section (u,
    v) in a MazeGraph instance
  task RegisterSection :
    (u: Int,  v: Int,
     n: Bool, e: Bool,
     s: Bool, w: Bool) -> (h: Unit) / write

  # Task to compute the next section (next_u,
    next_v) to go from (u,v) to the goal
  task CalcNextSection :
    (u: Int, v: Int, goal_u: Int, goal_v: Int)
      -> (next_u: Int, next_v: Int) / read
}
```

Figure 10. Material file for `MazeRunner`.

```
// Resource.c
struct MazeGraph { /* ... */ };

#define STACKSIZE_RegisterSection 2000
void RegisterSection(
  struct MazeGraph* w_res,
  int u, int v, int n, int e, int s, int w,
  int* h) {
  /* Set wall information */
}

#define STACKSIZE_CalcNextSection 2000
void CalcNextSection(
  const struct MazeGraph* r_res,
  int cur_u, int cur_v, int goal_u, int goal_v,
  int* next_u, int* next_v) {
  /* Search next direction */
}
```

Figure 11. Compilation results of task resources and tasks.

```
// MazeRunnerMain.c
void Input(...){ /* ... */ }
void Output(...){ /* ... */ }
int main(void){
  /* Initialization of sensors */
  /* Initialization of other devices */
  /* Initialization of resource instance (mg) */
  struct MazeGraph mg = { /* ... */ };
  ActivateMazeRunner(&mg);
  return 0;
}
```

Figure 12. Template I/O code for `MazeRunner`

but has not yet finished. `NotStarted` indicates that the task is not waiting for execution.

Nodes with `Future` type is defined in the syntax:

tasknode z:Future[$(\bar{\tau},\ldots,\bar{\tau})$] = $g(e_1,\ldots,e_n)$ **with** r **at** e_t

where z is node name, g is task name, r is task resource instance, $e_1,\cdots,e_n$ are task inputs, and e_t is issue condition. The task g is issued when z is not `Pending` and e_t is evaluated to `True`. When a task is issued, the input expressions (time-varying values) are stored as snapshots of their current values, and those snapshots are used during task execution.

Dependency analysis is not performed on **tasknode**s. The checking for task issue conditions and their issuing are performed after all normal node updating. There is no inconsistency due to dependencies in node definitions. We can refer to the previous value of a `Future` node by `@last` just like a normal node.

4.1.3 Limitations of tasknode.

There can only be at most one **tasknode** definition for a task in each task resource instance throughout the whole program. The compile time analysis checks this. This limitation allows the maximum number of tasks in the task execution queue to be determined statically. For example, for the `MazeRunner` module, the **tasknode** definition for the `RegisterSection` task that uses the task resource instance `mg` is on line 16, so a similar **tasknode** definition for the `RegisterSection` task is prohibited.

4.2 Runtime Extension

Emfrp is intended to run programs in small-scale resource-constrained embedded environments, especially on microcontrollers. Therefore, the following sections explain the runtime extensions for single-core microcontrollers. Designing an asynchronous task execution mechanism, we aimed (A1) to maintain consistency of task resources shared among tasks, (A2) to work in a small-scale environment such as a microcontroller, and (A3) not to require dynamic memory allocation (to the heap area).

4.2.1 Representing Tasks and Task Resources.

The task resources and tasks defined in Figure 10 are converted by the compiler into the template code shown in Figure 11. Task resources are converted to C structures, whose contents are completed by the user. Tasks are converted to C functions, which receive a pointer to a structure representing a task resource instance as their first argument. Each task function is expected to handle the task resource instance according to the **read** or **write** annotated to the task. A task function is executed under a particular static stack area of the size specified by the `STACKSIZE_XXX`. In order to determine the size of the allocated area, it is necessary to determine the

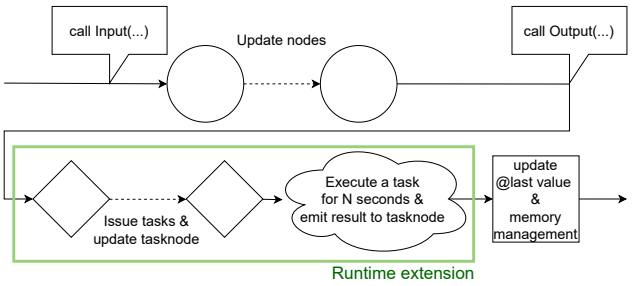

Figure 13. The flow of the extended iteration

Algorithm 1 Issue g^r (the task g with resource instance r) and update `tasknode` z

```
1:  function UPDATE_AND_ISSUE_g_WITH_r( )
2:      if g^r.pending then
3:          emit Pending to z
4:      else if eval(e_t) == true then
5:          v_1 ← eval(e_1); ... ; v_n ← eval(e_n)
6:          g^r.parameter ← (v_1, ..., v_n)
7:          if W_r ≠ ∅ ∧ MUTEX(g^r, E) then
8:              E.enqueue(g^r)
9:          else
10:             W_r.enqueue(g^r)
11:         end if
12:         emit Pending to z
13:         g^r.pending ← true
14:     else
15:         emit NotStarted to z
16:     end if
17: end function
```

maximum number of tasks that may be issued. For this reason, `tasknode` has a limit in the 4.1.3 section. This limitation and feature are made with (A3) in mind.

Figure 12 shows the template code for the main function and input/output functions generated from the top-level module. The difference from the conventional Emfrp template code is that the structure representing the task resource instance is defined as a local variable `mg`. The task resources specified as input to the top-level module are compiled into the arguments of the `ActivateMazeRunner` function. The user starts an Emfrp program by calling the `ActivateMazeRunner` function with `mg` as input.

4.2.2 Iterations.

Figure 13 shows the flow of the extended iteration. An asynchronous task execution phase is added to the iteration explained in section 2.4. Unfortunately, due to the task execution, the extended iteration has a longer execution time than the non-extended one. To deal with this, the user can define the task execution time to keep the responsiveness degradation within an acceptable level. During an iteration, tasks are executed only for the time specified by the user at compile time.

Algorithm 2 Asynchronous task execution

```
1:  function EXECUTE_TASK( )
2:      if E = ∅ then
3:          return
4:      end if
5:      g^r ← E.dequeue()
6:      (is_done, result) ← ctxsw_N(g, r, g^r.parameter, g^r.stack)
7:      if is_done then
8:          emit Ready(result) to z
9:          g^r.pending ← false
10:         while W_r ≠ ∅ ∧ MUTEX(W_r.peek(), E) do
11:             y^r ← W_r.dequeue()
12:             E.enqueue(y^r)
13:         end while
14:     else
15:         E.enqueue(g^r)
16:     end if
17: end function
```

4.2.3 Task Execution Mechanism.

Algorithm 1 and algorithm 2 are the core of our proposed asynchronous task execution mechanism. In our method, in order to achieve (A1), we use the `read` and `write` specified in the task definition as hints to the compiler and the runtime system. The runtime system has one execution queue E and waiting queues for each resource instance. W_r represents the queue for resource instance r. In each algorithm, g^r is an object representing a task g that manipulates a resource instance r. A task object in progress is contained in the execution list E. Task objects that have been issued but cannot be executed are enqueued in the waiting queue W_r. For a task object g^r, if (1) g is a `read` task and all tasks corresponding to r in E are `read` tasks, or (2) g is a `write` task and no task corresponding to r exists in E, g^r can be enqueued to E. Let MUTEX(g^r, E) be a predicate that determines this condition. Each algorithm uses MUTEX to control the mutual exclusion of task resource instances during task execution.

Algorithm 1 is a function that issues tasks. This function is executed for each g^r in the diamond part of Figure 13. In the algorithm, z, $e_1, \ldots, e_n$, e_t are from the `tasknode` definition: `tasknode` $z = g(e_1, \ldots, e_n)$ `with` r `at` e_t . eval(e) evaluates the time-varying value e. The variable g^r.pending indicates whether the status of the task object g^r is pending or not. The g^r.parameter is a variable that stores a snapshot of the task's arguments. If the task status is not PENDING, it checks the issue condition and issues the task. *MUTEX* is used to determine either E or W_r to insert the task object. The value of the `tasknode` is also updated.

Algorithm 2 is a function that intermittently executes the issued task. This function is executed in the cloud-shaped part of Figure 13. Task execution is round-robin, one per iteration. Task execution and pausing are handled using the operating system's context switch technique and the timer

interrupts. Line 6 of the algorithm is a pseudo code representing task execution by the context switch technique. The runtime system performs a context switch to task g for the time N given at compile time. At this time, a resource instance r and stored parameters (g^r.parameter) are passed to the task function. The task function runs under the statically allocated stack area (g^r.stack). After executing a task for a specified time, if the task is completed, the corresponding **tasknode** is overwritten with `Ready`, and other tasks of resource instance r are inserted into the execution queue E. Because task execution is round-robin and tasks are moved to the execution queue sequentially from the head of the waiting queue, this scheduling algorithm does not cause starvation (a state in which a task that has been issued remains unexecuted). Since this algorithm is simple and requires a timer interrupt (not special hardware), it can achieve (A2).

The number of task resource instances and the number of tasks can be determined statically due to the restrictions of the **tasknode** definition (section 4.1.2). Based on this information, the maximum lengths of the execution and waiting queues can be determined at compile time. Thus, (A3) can be achieved.

Note that asynchronous tasks are functions written in the C language. Therefore, there is no guarantee of memory safety by Emfrp. The memory provided for task execution may be insufficient for the stack area, which may cause unexpected bugs during program execution. External tools such as StackAnalyzer [4] should be used to guarantee safety to prevent this.

4.3 Example in Extended Emfrp

This section explains the behavior of the `MazeRunner` module in Figure 9. First, it is explicitly specified in line 3 that `mg` is used as the instance of the task resource `MazeGraph`. The other input/output nodes are almost the same as those of the non-extended `MazeRunnerEmfrp` module. However, since the recording of wall information and the searching next target section is now handled within the module, they are not contained in the input/output nodes. In line 9, the `MovePID` module is expanded as a sub-module. Hence, the robot moves in response to changes of the target section (`to_u`, `to_v`).

`near_target` node on line 12 becomes true when the center of the robot is sufficiently within the target section. By detecting the rising edge of this node, the wall information recording task `RegisterSection` on lines 15–17 is issued. When completed, this task returns a value of type `Unit`. `wait` function defined in lines 9–11 of Figure 10, detects the completion of the computation of a `Future` node as a rising edge. In lines 20–21, a maze search task (`CalcNextSection`) is issued using this function after the wall information has been recorded. In other words, when the robot is close enough to the target section, the wall information of the target section is recorded, and the search for the next target section begins. Since `MovePID` module continues to work during this search, the reactive behavior does not stop. The pattern match in lines 41 and 42 detects the completion of `CalcNextSection` task. Line 41 means the task was completed after the robot reached the target section. In this case, the new target section is immediately updated in (`to_u`, `to_v`). Line 42 means the task was completed before the robot reached the target section. The new target section is temporarily moved to `tmp` node. After that, when the robot reaches the target, it matches line 43, so the stored `tmp` (the new target section) is put into (`to_u`, `to_v`).

When the new target section is set, `near_terget` node becomes false. When the robot gets close enough to the new target, `near_terget` node becomes true, and a rising edge is triggered. Thus, the above steps are repeated until the robot reaches the goal.

From the above, the improved exploration was implemented in the `MazeRunner` module. In addition, although the program does not explicitly show the data structure implementation subject to the heavy task, it shows which resource (data structure) is manipulated via the task resource instance. The problem (P2) has been relaxed, and the `MazeRunner` module is a relatively reader-friendly program.

5 Related Works

The *future* (or *promise*) pattern is a design pattern in concurrent programming that delays the fetching of the result of an asynchronous calculation until the value of the calculation is needed. Java, JavaScript, Rust, Scala, and many other languages that support asynchronous execution implement this pattern [5, 9, 10, 18]. This method is typically used for pipelining asynchronous computational processes. In this study, heavy task execution is indeed an asynchronous process. The necessity of managing and utilizing the computation state of issued tasks led to the introduction of the future pattern.

In reactive programming [3], reactive behaviors are represented as data flows in a dependency graph between time-varying values. Thus, how to represent behaviors that are difficult to handle as data flows, such as heavy tasks in this study, is a language design problem. Van den Vonder et al. analyzed this problem for reactive programming libraries based on imperative languages (e.g., REScala [22], ReactJS [14], RxJS [25]) [26]. As a result, they advocated the *Reactive Thread Hijacking Problem (RTHP)*. They also proposed the *Actor-Reactor model* [26], a method to separate and modularize the reactive and procedural behavior descriptions. This model categorizes time-consuming processes (heavy task) and side effects as *actors* [1] with procedural descriptions. They are separated from reactive behavior descriptions (*reactors*), represented as dataflow graphs.

[4] https://www.absint.com/stackanalyzer/index.htm

In the Actor-Reactor model, the coordination between actors and reactors works by feedback from the reactor's output through the actor to the reactor's input. They use Stella, a language designed for Actor-Reactor models, to write actors and reactors in a single language. As a result, the dependencies between actors and reactors are explicitly defined. So the readability and maintainability of the program are maintained. On the other hand, when combining Emfrp with the Actor-Reactor model, the actor is an input/output function written in the C language. Thus, the implicit dependencies between output and input nodes across two languages, discussed in section 3.3.2 (P2), reduce the readability and maintainability of the program. In this study, we solve this problem by introducing **`tasknode`** definition and `Future` type.

Lustre [11, 16] is a synchronous dataflow language that allows programming using the `fby` operator, similar to Emfrp's `@last`. Cohen et al. introduced *future* to Lustre, which enables concurrency, pipelining, and jitter control [4]. Their future allows Lustre code to be processed concurrently with other Lustre codes. They formally prove that the semantics are preserved even when removing the future from a code with futures. On the other hand, our proposed method introduces **`tasknode`**s and `Future`s to achieve (time-consuming) foreign function calls with no loss of responsiveness. It does not provide for parallel processing between Emfrp codes. Since the computation performed by **`tasknode`**s is difficult to express in non-extended Emfrp, the preservation of semantics requires a different discussion than that of Cohen et al.

Hailstorm [23] is a functional language for IoT applications inspired by the Arrowized FRP [15, 17]. Users write programs by composing signal functions using `&&&` and `>>>` operators specific to Arrowized FRP. *Juniper* [12] is an FRP language for Arduino [2], a small microcontroller. The `foldp` function in Juniper (originated from Elm [7]) allows the temporal accumulation of time-varying values. Although these languages and *Lustre* (described above) have different notation schemes, time-varying values are updated sequentially in a single main thread, as in Emfrp. Therefore, RTHP reduces the responsiveness of the whole system. We expect that our runtime system, designed based on approaches (A1), (A2), and (A3), will help improve this issue.

6 Conclusions and Future Tasks

We designed an asynchronous task execution mechanism for Emfrp, an FRP language for small-scale embedded systems, that allows reactive behavior and relatively time-consuming task coordination. As language extensions, we introduced the definition of **`tasknode`** and `Future` types to issue asynchronous tasks and obtain their results. As a runtime extension, we proposed an asynchronous execution mechanism that can work even in a resource-constrained environment by inserting a task execution phase after the time-varying value update.

The proposed method treats all resources manipulated by a task as task resources. By splitting task resources into smaller resources and merging and sharing them among tasks, more fine-grained exclusion control and resource management are possible. The design of the execution mechanism and task execution priorities to achieve this is a future issue.

Future tasks also include the implementation of the proposed method and the measurement of the system responsiveness. The latter contains the time overhead of heavy task execution, and the spatial overhead of the run-time memory required for asynchronous task execution.

Acknowledgments

This work was supported in part by JST SPRING, Grant Number JPMJSP2106 and JSPS KAKENHI Grant Numbers 21K11822 and 22K11967.

References

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

[2] Arduino. Accessed Sep. 2022. Arduino. https://www.arduino.cc/

[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] Albert Cohen, Léonard Gérard, and Marc Pouzet. 2012. Programming Parallelism with Futures in Lustre. In *Proceedings of the Tenth ACM International Conference on Embedded Software* (Tampere, Finland) *(EMSOFT '12)*. Association for Computing Machinery, New York, NY, USA, 197–206. https://doi.org/10.1145/2380356.2380394

[5] Oracle Corporation. Accessed Sep. 2022. Future (Java Platform SE 8). https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html

[6] Antony Courtney, Henrik Nilsson, and John Peterson. 2003. The Yampa Arcade. In *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell* (Uppsala, Sweden) *(Haskell '03)*. ACM, 7–18. https://doi.org/10.1145/871895.871897

[7] Evan Czaplicki and Stephen Chong. 2013. Asynchronous Functional Reactive Programming for GUIs. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*. ACM, 411–422. https://doi.org/10.1145/2499370.2462161

[8] 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

[9] Mozilla Foundation. Accessed Sep. 2022. Promise - JavaScript | MDN. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[10] Rust Foundation. Accessed Sep. 2022. Future in std::future - Rust. https://doc.rust-lang.org/std/future/trait.Future.html

[11] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. 1991. The synchronous data flow programming language LUSTRE. *Proc. IEEE* 79, 9 (1991), 1305–1320.

[12] Caleb Helbling and Samuel Z Guyer. 2016. Juniper: A Functional Reactive Programming Language for the Arduino. In *4th International Workshop on Functional Art, Music, Modelling, and Design (FARM 2016)*. ACM, 8–16. https://doi.org/10.1145/2975980.2975982

[13] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. 2003. Arrows, Robots, and Functional Reactive Programming. In *Advanced Functional Programming*. Lecture Notes in Computer Science, Vol. 2638. Springer-Verlag, 159–187. https://doi.org/10.1007/978-3-540-44833-4_6

[14] Facebook Inc. Accessed Sep. 2022. ReactJS: A JavaScript library for building user interfaces. https://reactjs.org

[15] Hai Liu, Eric Cheng, and Paul Hudak. 2009. Causal Commutative Arrows and Their Optimization. *SIGPLAN Not.* 44, 9 (aug 2009), 35–46. https://doi.org/10.1145/1631687.1596559

[16] LustreV6 2020. *Verimag Lustre V6.* Retrieved Sep 13, 2022 from http://www-verimag.imag.fr/Lustre-V6.html

[17] Henrik Nilsson, Antony Courtney, and John Peterson. 2002. Functional Reactive Programming, Continued. In *Proceedings of the 2002 ACM SIGPLAN Workshop on Haskell* (Pittsburgh, Pennsylvania) *(Haskell '02)*. ACM, 51–64. https://doi.org/10.1145/581690.581695

[18] Programming Methods Laboratory of École Polytechnique Fédérale de Lausanne. Accessed Sep. 2022. Futures and Promises | Scala Documentation. https://docs.scala-lang.org/overviews/core/futures.html

[19] Izzet Pembeci, Henrik Nilsson, and Gregory Hager. 2002. Functional Reactive Robotics: An Exercise in Principled Integration of Domain-Specific Languages. In *4th International Conrefernce on Principles and Practice of Declarative Programming (PPDP 2002)*. ACM, 168–179. https://doi.org/10.1145/571157.571174

[20] Yoshitaka Sakurai, Sosuke Moriguchi, and Takuo Watanabe. 2021. Functional Reactive Programming for Embedded Systems with GPGPUs. In *10th International Conference on Software and Computer Applications (ICSCA '21)*. ACM, 75–80. https://doi.org/10.1145/3457784.3457795

[21] Yoshitaka Sakurai and Takuo Watanabe. 2019. Towards a Statically Scheduled Parallel Execution of an FRP Language for Embedded Systems. In *6th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems (REBLS 2019)*. ACM, 11–20. https://doi.org/10.1145/3358503.3361276

[22] 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

[23] Abhiroop Sarkar and Mary Sheeran. 2020. Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications. In *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming* (Bologna, Italy) *(PPDP '20)*. ACM, Article 12, 16 pages. https://doi.org/10.1145/3414080.3414092

[24] 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

[25] RxJS Team. Accessed Sep. 2022. RxJS: Reactive Extensions Library for JavaScript. https://rxjs.dev

[26] Sam Van den Vonder, Thierry Renaux, Bjarno Oeyen, Joeri De Koster, and Wolfgang De Meuter. 2020. Tackling the Awkward Squad for Reactive Programming: The Actor-Reactor Model. In *34th European Conference on Object-Oriented Programming (ECOOP 2020) (Leibniz International Proceedings in Informatics (LIPIcs), Vol. 166)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 19:1–19:29. https://doi.org/10.4230/LIPIcs.ECOOP.2020.19

[27] 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

[28] Akihiko Yokoyama, Sosuke Moriguchi, and Takuo Watanabe. 2021. A Functional Reactive Programming Language for Small-Scale Embedded Systems with Recursive Data Types. *Journal of Information Processing* 29 (Oct. 2021), 685–706. https://doi.org/10.2197/ipsjjip.29.685

Received 2022-09-13; accepted 2022-10-10