Event by Timing: Periodic and Time-Sequencing Responses

Sosuke Moriguchi* and Takuo Watanabe

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

Abstract. Functional reactive programming (FRP in short) abstracts values that change over time as time-varying values and the timing of responses as events. Using these abstractions, reactive systems are described as data flows between time-varying values and the occurrence of events. In this paper, we propose TEFRP, a functional reactive programming language for embedded systems in which the data flow of time-varying values can be switched at each periodic timing. The ability to switch the update method at each time facilitates the description of systems in which the responses change over time. Such systems include, for example, programs that display received data by turning LEDs on and off, and communication protocols that perform fixed pre-processing and post-processing. TEFRP treats the time from the start of execution as an event. Although events in typical FRP are assumed not to occur at the same time, time descriptions may include the same timing. TEFRP allows logical combinations of periodic timings and actively uses such overlaps to improve descriptiveness. This paper also discusses methods of converting logical combinations to simpler forms.

Keywords: functional reactive programming, periodic task, event-based programming

1 INTRODUCTION

A reactive system returns responses to periodic or interrupted inputs. This reaction can be an immediate response to an input, a grouping of multiple inputs, or a sequential response to an input. Reactive systems can be implemented by polling, callbacks, or setting up interrupt handlers. However, these techniques complicate the control flow of the system and fragment the description of the tasks performed at each hour. Functional reactive programming (FRP in short) is a technique in which the control flow for response is derived from the data flow. FRP has a wide range of applications, including animation, GUI, robotics, and embedded systems [3, 2, 9, 8, 4].

In FRP, values that change with time, such as inputs, outputs, and internal states, are abstracted as time-varying values. By describing the data flow of time-varying values in a functional manner, a computation can be constructed to reflect changes occurring in the inputs to the outputs. In other words, FRP makes it possible to construct a system that guarantees responsiveness without specifying complex control flows.

Time-varying values represent continuously changing values, but in an actual system, changes occur only at specific times. Events are used to represent the timing of such changes and responses. Time-varying values are processed in response to the handling of each event. FRP, where events are used explicitly, assumes that events do not occur at the same time. To avoid glitches, which are data races on time-varying values, when two events occur at the same time in an actual system, they are decomposed into two events and updated sequentially.

If there are input and output devices with different periods, the events driving each device are created independently. Consider the case where the value from an input is divided and used for the output. Assume that the number of output divisions is constant for one reading of the input. In this case, the events for the input and the events for the output must be synchronized in terms of their backward/forward relationship and period. However, since they are generated independently, there is a possibility that the next input may be processed without outputting the expected number of times due to the accumulation of slight deviations (jitter), or conversely, the next output may be processed before the next input is received. To ensure that the number of outputs is maintained, it is necessary to generate events with time shifts so that the cycles do not overlap, and to synchronize the shift in the generation time of each event. This method, however, requires consideration of whether or not the same problem will occur for each existing device when another device is added. In an environment where events are fired internally within a program, it is easy to synchronize the timing of event generation, thus avoiding the problem. However, when the number of kinds of events increases, the order of firing must be determined based on the dependencies among time-varying values.

In this paper, we propose an FRP language, TEFRP, which assumes that events are generated by time from the start of the system. Hereafter, events that occur due to the passage of time are called time events, and a description that refers to one or more time events is called a time description. TEFRP is based on Event-driven FRP [16] and Emfrp [13], and describes an update process for each time description in each time-varying value. This mechanism can be applied to the description of sequential responses that are output sequentially at each time, in addition to considering input/output cycles. Different time descriptions may refer to the time events occurring at the same time, but in contrast to asynchronous events in most FRPs, these events represent the same event. Taking advantage of this, time descriptions in TEFRP use logical combinations to achieve flexibility.

The contributions of this paper are as follows

– We propose a mechanism to switch to a different update process for each time event. While the entire reactive system can be described declaratively, it is also easy to describe a system that shows sequential responses.

– By specifying the update timing, we provide a mechanism to check whether the access to the current or previous value is valid. This ensures that appropriate initialization is performed without using dummy values.
– We provide a method for converting a time description written using logical combination into a simplified form. The converted form consists only of constant time or periodic timing, and there is no logical combination at all. This conversion allows for easy scheduling while ensuring the descriptiveness of the surface language.

The system assumed for TEFRP in this paper is a soft real-time embedded system, and the time required to process each time-varying value update is assumed to be sufficiently short relative to the overall cycle. Therefore, the scheduling of tasks is assumed to be simple, with all tasks having the same priority.

The paper is organized as follows. In Section 2 we describe a motivating example, and in Section 3 we describe this example in TEFRP. In Section 4 we discuss the syntax, semantics, and limitations of TEFRP. Section 5 discusses methods for converting definitions into equivalent, simple programs. Comparisons with related studies are made in Section 6 and the paper is summarized in Section 7.

2 MOTIVATION

Using an existing reactive programming library, REScala [11], we describe an example program using time events and present its problems. Here, we consider an example that outputs an input value as a bit-by-bit true/false value. This output value is used, for example, to correspond to the ON/OFF of an LED. For simplicity, let the input value be an 8-bit integer and read every 5 s. Also, we assume that the output is performed 8 times every 500 ms, starting with the lower to upper bits, and then `false` is performed as a 1-second interval. Such a system is described as Listing 1.

The input is held in signal `input` through event `e1`. The signal `bit` is doubled each time an event `e2` fires, thereby changing the position of the filter. The output is represented by the signal `output`, which changes along with the change in `bit`. However, if event `e3` is fired with `true`, `bit` is set to be `false`. This expresses the state after all bits have been output.

In this program, the output changes as `e2` or `e3` fires. The reason that the firing of `e3` in line 17 is after the firing of `e2` is that if it fires before this, `output` will instantaneously become `true` if the most significant bit of the input is 1 and the least significant bit is 0. When the value output externally is used to control a physical device such as a motor, such an instantaneous change in value may cause abnormal actuation or degradation. The firing order of events is adjusted to prevent this, but sufficient care must be taken regarding the order when multiple events are desired to occur simultaneously. In addition, when an output is changed by multiple events, it becomes difficult to follow at what time the change occurs.

```
// reactive part
val e1 = Evt[Int]() // input event
val input: Signal[Int] = e1.latest(0)
val e2 = Evt[Unit]() // output event
val f = (x:Int) => if (x == 128) 1 else (x*2)
val bit = e2.iterate(128)(f)
val e3 = Evt[Bool]() // waiting
val wait = e3.latest(true)
val output = Signal {
    !wait && (input.now / (bit())) % 2 > 0
}
// event part
while (true) {
    Thread sleep 1000
    e1.fire(sense()) // get input
    e2.fire() // 1st bit
    e3.fire(false)
    Thread sleep 500
    for(_ <- 2 to 8) { // 2nd to 8th bit
        e2.fire()
        Thread sleep 500
    }
    e3.fire(true)
}
```

Listing 1. Bitwise integer output in REScala

3 TIMING AS AN EVENT

This section describes the example in the previous section written in TEFRP. The formal grammar and other details are discussed in the next section. The program described is like Listing 2.

```
module Bitwise
  in  input { 5s * @n + 1s } : Int
  out output { 500ms * @n + 1s
               but not(5s * @n + 5500ms) } : Bool

node bit : Int = {
    5s * @n + 1s => 1;
    500ms * @n + 1500ms
      but not(5s * @n + 5s or 5s * @n + 5500ms)
        => bit@last * 2;
}
node output : Bool = {
    5s * @n + 5s => false;
    500ms * @n + 1s but not(5s * @n + 5500ms)
      => (input / bit) % 2 > 0;
}
```

Listing 2. Bitwise output in TEFRP

The structure of the TEFPR program is similar to that of Emfrp [13], but the update expressions are written for each time event, as in Event-driven FRP [16]. Like Emfrp, TEFRP also declares time-varying values with the keyword `node`.

Therefore, from now on in this paper, time-varying values in TEFRP will be referred to as *node*.

A program is composed of modules (`module`), which declare input nodes (`in`) and output nodes (`out`). Input and output nodes contain an *time description* that represents the time event at which the update is to take place. For example, the `input` node (line 2) declares a time description that represents reading from after start 1 s (`1s`) to after 5 s (`5s * @n`). Note that `@n` used in the time description is interpreted as an integer greater than or equal to 0. Time descriptions in TEFRP are not limited to such simple cycles, but can also include the logical conjunction and disjunction of multiple descriptions, as well as the description of exclusions. The output, `output` node (lines 3–4), is updated every 500 ms starting from the start1 s, but not every 5 s starting from the5500 ms (`but not(...)`)[1].

Lines 6 through 11 and 12 through 16 describe the definitions of `bit` and `output`, respectively. Both definitions have two types of time descriptions. If multiple time descriptions are written, the update process is performed according to the first-match policy. For example, in the definition of `output`, for time events that start at 5 s and after every 5 s, the second description (line 14) also matches, but TEFRP will use the first description (line 13), which makes the value of `output` be `false`.

The `bit@last` in line 10 refers to the result of updating `bit` in the previous time event. This operator can be used not only for the time-varying value that is about to be updated but also for other time-varying values. However, two conditions are necessary for the use of `@last`. One is that the target node must be updated at the time event being updated so that it makes sense to read one previous update. The other is that one previous update must exist so that a valid value can be obtained. Both can be easily checked with the SMT solver since the conditions can be expressed in the form of integer linear constraints based on the time description. Similarly, the combination of the time description used in the output node (lines 2–3) and the time description used in the node declaration (lines 13–14) is equivalent, which is also expressed as an integer linear constraint. The detailed constraint expressions are described in Section 4.3.

4 TEFRP

This section provides a formal discussion of the syntax and the semantics of TEFRP.

4.1 Syntax

Figure 1 shows the syntax of TEFRP.

T is a metavariable that represents a time description. T can express a description that directly specifies a time or periodic timing (AT), as well as a

[1] Note that the time descriptions in the output nodes of the module are redundant and will be omitted when the processor is implemented

$$\begin{array}{lll}
C & : & \text{Constant integer values}\\
ID & : & \text{Identifier}\\
M & ::= & \texttt{module in } N^* \texttt{ out } N^* \ D\\
N & ::= & ID\texttt{\{ } T \texttt{ \} : } Type\\
D & ::= & \texttt{node } ID \texttt{ : } Type \texttt{ = \{ } U^+ \texttt{ \}}\\
U & ::= & T \texttt{ => } E\texttt{;}\\
T & ::= & AT \mid T \texttt{ or } T \mid T \texttt{ and } T \mid T \texttt{ but not(}T\texttt{)}\\
AT & ::= & CT \mid CT\texttt{*@n+}CT\\
CT & ::= & C\texttt{ us} \mid C \texttt{ ms} \mid C \texttt{ s}\\
E & ::= & C \mid ID \mid ID\texttt{@last}\\
 & \mid & \ldots
\end{array}$$

Fig. 1. The syntax of TEFRP

combination of the two by logical combination. Periodic timing is described as a linear expression with one variable `@n` as an integer greater than or equal to 0. In the logical combination, `not` is allowed only in conjunction in a way that restricts the timing of something, rather than at an arbitrary point. It represents an exception to the underlying timing but does not represent all other (possibly continuous) times.

The module has several constraints.

1. A reference to another node in an update expression at each node is called a dependency. Dependencies between nodes must not be circular.
2. To reference a node, the referenced node must have been updated at least once at the same time or earlier.
3. To use the `@last` operator to refer to the previous value (the value before the update) of a node, the referenced node must have been updated at the same time. Also, it must have been updated at least once before that for ensuring existence of the previous value.
4. The time description described as an output node in the module must represent the same time event as the combination of time descriptions used in the node definition.

The first constraint is verified by performing a topological sort of the reference relationships of the nodes in the program as an ordinal relationship. By using the order obtained in this way as the update order of the nodes, it can be guaranteed that the dependent node is always updated before the dependent source is updated. The second to fourth constraints are discussed in Section 4.3 because they involve the semantics of the time description.

4.2 Semantics

First, we define the semantics for time descriptions. The semantics of a time description is expressed in terms of a set of times. In the semantics defined here, all times are expressed in microseconds (`us`), and those using `ms` or `s` are converted

$$
\begin{array}{ll}
[\![C\ \texttt{us}]\!]_{\mathrm{C}} & = C \\
[\![C\ \texttt{ms}]\!]_{\mathrm{C}} & = C \times 1000 \\
[\![C\ \texttt{s}]\!]_{\mathrm{C}} & = C \times 1000000 \\
[\![CT]\!]_{\mathrm{A}} & = \{[\![CT]\!]_{\mathrm{C}}\} \\
[\![CT_1\texttt{*@n+}CT_2]\!]_{\mathrm{A}} & = \{C_2, C_1 + C_2, C_1 \times 2 + C_2, \ldots\} \\
 & \quad \text{where } C_1 = [\![CT_1]\!]_{\mathrm{C}},\ C_2 = [\![CT_2]\!]_{\mathrm{C}} \\
[\![AT]\!]_{\mathrm{T}} & = [\![AT]\!]_{\mathrm{A}} \\
[\![T_1\ \texttt{or}\ T_2]\!]_{\mathrm{T}} & = [\![T_1]\!]_{\mathrm{T}} \cup [\![T_2]\!]_{\mathrm{T}} \\
[\![T_1\ \texttt{and}\ T_2]\!]_{\mathrm{T}} & = [\![T_1]\!]_{\mathrm{T}} \cap [\![T_2]\!]_{\mathrm{T}} \\
[\![T_1\ \texttt{but not(}T_2\texttt{)}]\!]_{\mathrm{T}} & = [\![T_1]\!]_{\mathrm{T}} \setminus [\![T_2]\!]_{\mathrm{T}}
\end{array}
$$

Fig. 2. The semantics of time descriptions

Algorithm 1 The semantics for a node definition (*Us* is a list of update descriptions)

```
1: function node_behavior($Us$)
2:     $Result = \emptyset$                          ▷ A set of pairs of time and update expressions
3:     $Used = \emptyset$                            ▷ A set of time already allocated in $Result$
4:     foreach $T$ => $E \in Us$ do
5:         $Ts = [\![T]\!]_{\mathrm{T}} \setminus Used$
6:         $Result = Result \cup \mathrm{map}(t \mapsto (t, E), Ts)$
7:         $Used = Used \cup Ts$
8:     end for
9:     return $Result$
10: end function
```

to constant times 1000 or 1 million, respectively. The function representing the semantics of the time description, shown in Fig. 2, has a different subscript for each syntax.

The update process of a node is performed by computing the corresponding update expression at the time obtained from the semantics of the time description. The set of time and expression pairs is defined as the node's behavior. The algorithm to obtain the behavior from the update expressions is as shown in Algorithm 1. Considering the first-match policy, we add the semantics of the previous time description, excluding the time whose behavior is already determined from the semantics of the previous time description.

The semantics of expressions are omitted in this paper because they are straightforward. The module operates by determining the order of nodes based on their dependencies and performing update processing according to the behavior of each node over time. As with Emfrp, the input is implemented in the form of externally defined functions that are called by the timing described in TEFRP.

4.3 Constraints for Modules

Here we describe the constraints on modules listed in the section 4.1. The constraints of a time description in a module are in the form of whether any given time represented by the time description is in a (dis)equal relationship with any of the times represented by the other time descriptions. Therefore, we define the predicate P that represents the existence of a time in the set of times represented by time description T that is in binary relation R with time t, as follows.

$$\begin{aligned}
P(CT, t, R) &= [\![CT]\!]_{\mathrm{C}}\; R\; t \\
P(CT_1\texttt{*@n+}CT_2, t, R) &= \exists n. [\![CT_1]\!]_{\mathrm{C}} * n + [\![CT_2]\!]_{\mathrm{C}}\; R\; t \\
P(T_1 \texttt{ or } T_2, t, R) &= P(T_1, t, R) \lor P(T_2, t, R) \\
P(T_1 \texttt{ and } T_2, t, R) &= P(T_1, t, R) \land P(T_2, t, R) \\
P(T_1 \texttt{ but not}(T_2), t, R) &= P(T_1, t, R) \land \neg P(T_2, t, R)
\end{aligned}$$

Assume the following construction for node `n`, which refers to node `m`. Note that for the discussion on the previous value, we assume that the `m` in line 5 is `m@last`.

```
1  node n = {                        node m = {
2    $T_1$ = ⋯;                         $T'_1$ = ⋯;
3    ⋯;                                 ⋯;
4    $T_x$ = ⋯;                         $T'_y$ = ⋯;
5    $T$ = ⋯ m ⋯;                     }
6    ⋯;
7  }
```

Consider the case of a reference to node `m` for line 5 of this `n`. This update description is chosen at time t, which means that t does not match any of T_1 to T_x, but T does. Thus, this is represented by $\bigwedge \neg P(T_i, t, =) \land P(T, t, =)$. Since we only need to guarantee that `m` is updated in the past or at the same time under these conditions, we only need t to match any of T'_1 to T'_y. In other words, the whole is represented by $\forall t. (\bigwedge \neg P(T_i, t, =) \land P(T, t, =)) \Rightarrow \bigvee P(T'_j, t, \leq)$. Similarly, it is sufficient to inspect the conditions on the previous value, one whose consequent part is $\bigvee P(T'_j, t, =)$ and the other whose consequent part is $\bigvee P(T'_j, t, <)$. Since these are in the category of Presburger arithmetic and the quantifiers are hardly nested, it is possible to decide its validity in a reasonable time.

5 Serializing time descriptions

Because of the mixture of various time descriptions in the node definitions, it is not so easy to know when the next update at a given point in time will be, and by which expression. Since TEFRP is intended for embedded systems, it is not appropriate to use complex calculations to determine the next time. Here, we give a method, *serialization*, to convert a node definition containing an arbitrary time description into a definition with simple time descriptions that behave in the same way. A simple time description here consists of AT alone in Figure 1.

Algorithm 2 Serialized phase and period

```
1:  function timing($T$)
2:      if $T = CT$ then
3:          return $([\![CT]\!]_C, 1)$
4:      else if $T = CT_1$* @n +$CT_2$ then
5:          return $([\![CT_2]\!]_C, [\![CT_1]\!]_C)$
6:      else if $T = T_1$ or $T_2$ or $T_1$ and $T_2$ or $T_1$ but not($T_2$) then
7:          $(phase_1, period_1) = \text{timing}(T_1)$
8:          $(phase_2, period_2) = \text{timing}(T_2)$
9:          return $(\max(phase_1, phase_2), \text{lcm}(period_1, period_2))$
10:     end if
11: end function
12:
13: function max_timing($Us$)
14:     $period = 1$
15:     $phase = 0$
16:     foreach $T$ => $E \in Us$ do
17:         $(phase_T, period_T) = \text{timing}(T)$
18:         $period = \text{lcm}(period, period_T)$
19:         $phase = \max(phase, phase_T)$
20:     end for
21:     return $(phase, period)$
22: end function
```

The entire periodic time descriptions behave cyclically by the least common multiplier of their respective periods. Therefore, the result of serialization is a set of copies of each update description for each phase, with the period fixed to the least common multiplier. This ensures that the behavior is identical to the meaning of the original update description

In Algorithm 2, the phase until the periodic behavior is shown, and the period at that time is calculated. Then, in Algorithm 3, the update descriptions with constant times before the periodic behavior and the time events that started in the earlier phases before the periodic behavior is entered are expanded. Similarly, in Algorithm 4, the phases of the update descriptions performed in the periodic behavior are expanded. Finally, Algorithm 5 generates the serialized update descriptions from the phases generated by the above algorithms. To realize the first-match policy, the phases that have already been generated are kept (lines 9 and 12), and they are excluded before generating corresponding update descriptions (lines 7 and 10).

We describe these algorithms using `bit` in Listing 2 as an example. From now on, for the sake of readability, the values returned by each algorithm will be described as time instead of numerical values representing microseconds. For `5s * @n + 1s` and `500ms * @n + 1s but not(5s * @n + 5s or 5s * @n + 5500ms)`, the timing function in Algorithm 2 returns (5 s, 1 s), (5 s, 5500 ms) respectively.

The constants function of Algorithm 3 is applied with this result. The function enumerates the time that occurs before 5500 ms, so applying it to `5s * @n`

Algorithm 3 Constant phase extraction

```
function constants($phase, T$)
    $result = \emptyset$
    if $T = CT$ then
        $result = \{[\![CT]\!]_C\}$
    else if $T = CT_1$* @n +$CT_2$ then
        $c = [\![CT_2]\!]_C$
        while $c \leq phase$ do                ▷ Enumerate time that occurs before $phase$
            $result = result \cup \{c\}$
            $c = c + [\![CT_1]\!]_C$
        end while
    else if $T = T_1$ or $T_2$ then
        $result = \mathrm{constants}(phase, T_1) \cup \mathrm{constants}(phase, T_2)$
    else if $T = T_1$ and $T_2$ then
        $result = \mathrm{constants}(phase, T_1) \cap \mathrm{constants}(phase, T_2)$
    else if $T = T_1$ but not($T_2$) then
        $result = \mathrm{constants}(phase, T_1) \setminus \mathrm{constants}(phase, T_2)$
    end if
    return $result$
end function
```

Algorithm 4 Periodic phase extraction

```
function periodic($phase, period, T$)
    $result = \emptyset$
    if $T = CT_1$* @n +$CT_2$ then
        $c = [\![CT_2]\!]_C$
        while $c \leq phase$ do                      ▷ Calculate the first time *after* $phase$
            $c = c + [\![CT_1]\!]_C$
        end while
        while $c \leq phase + period$ do   ▷ Enumerate time that occurs in one $period$
            $result = result \cup \{c\}$
            $c = c + [\![CT_1]\!]_C$
        end while
    else if $T = T_1$ or $T_2$ then
        $result = \mathrm{periodic}(phase, period, T_1) \cup \mathrm{periodic}(phase, period,_2)$
    else if $T = T_1$ and $T_2$ then
        $result = \mathrm{periodic}(phase, period, T_1) \cap \mathrm{periodic}(phase, period, T_2)$
    else if $T = T_1$ but not($T_2$) then
        $result = \mathrm{periodic}(phase, period, T_1) \setminus \mathrm{periodic}(phase, period, T_2)$
    end if
    return $result$
end function
```

Algorithm 5 Serializing node definition

```
 1: function normalize($Us$)
 2:     $(phase, period) = \text{max\_timing}(Us)$
 3:     $result = [\,]$
 4:     $usedConsts = \emptyset$                ▷ A set of phases allocated as constant timing
 5:     $usedPhases = \emptyset$                ▷ A set of phases allocated as periodic timing
 6:     foreach $T$ => $E \in Us$ do
 7:         $consts = \text{constants}(phase, T) \setminus usedConsts$
 8:         $result = result \mathbin{++} \text{mapL}(p \mapsto (p$ us => $E), consts)$
 9:         $usedConsts = usedConsts \cup consts$
10:         $phases = \text{periodic}(phase, period, T) \setminus usedPhases$
11:         $result = result \mathbin{++} \text{mapL}(p \mapsto (period$ us*@n+$p$ us => $E), phases)$
12:         $usedPhases = usedPhases \cup phases$
13:     end for
14:     return $result$
15: end function
```

`+ 1s` returns only 1 s, while applying it to `500ms * @n + 1s but not(5s * @n + 5s or 5s * @n + 5500ms)` returns 1 s, 1500 ms, 2 s, 2500 ms, 3 s, 3500 ms, 4 s and 4500 ms. The reason why 5 s and 5500 ms are not included in the latter is that they are excluded from the whole because the description under `but not` returns these times. With these results, in lines 7–9 of the Algorithm 5, 1 s is chosen from the former, and the rest from the latter.

Similarly, Algorithm 4 enumerates the times that occur between 5500 ms and 10 500 ms. It returns 6 s for the former and 6 s, 6500 ms, 7 s, 7500 ms, 8 s, 8500 ms, 9 s and 9500 ms for the latter. In lines 10–12 of Algorithm 5, 6 s is selected from the former and the rest from the latter.

By serializing whole of Listing 2, both `bit` and `output` are expanded in period 5 s and phase to period 5500 ms, as shown in Listing 3. Note that Algorithm 5 converts everything to microseconds, but here it is written in seconds or milliseconds for readability.

The serialization procedure allows all time descriptions to be expressed either in constant times with no period or in time descriptions with the same period but different only in phases. As a result, the obtained time descriptions do not overlap and are not affected by the first-match policy, so it is permissible to reorder the sequence phase by phase. This means that the next time event to be updated can be easily computed.

The serialized definitions produce the same set in terms of the behavior defined by Algorithm 1. The *usedConsts* and *usedPhases* in Algorithm 5 correspond to *Used* in Algorithm 1. Therefore, before and after the serialization of a single time description, the behavior and the time of exclusion are preserved. This shows that the overall behavior is preserved. The formal proof is a subject for future work.

```
node bit : Int = {
 1s => 1;                           1500ms => bit@last * 2;
 2000ms => bit@last * 2;            2500ms => bit@last * 2;
 3000ms => bit@last * 2;            3500ms => bit@last * 2;
 4000ms => bit@last * 2;            4500ms => bit@last * 2;
 5s * @n + 6s => 1;                 5s * @n + 6500ms => bit@last * 2;
 5s * @n + 7000ms => bit@last * 2; 5s * @n + 7500ms => bit@last * 2;
 5s * @n + 8000ms => bit@last * 2; 5s * @n + 8500ms => bit@last * 2;
 5s * @n + 9000ms => bit@last * 2; 5s * @n + 9500ms => bit@last * 2;
}
node output : Bool = {
 5s => false;                 1s => (input / bit) % 2 > 0;
 1500ms => (input / bit) % 2 > 0; 2000ms => (input / bit) % 2 > 0;
 2500ms => (input / bit) % 2 > 0; 3000ms => (input / bit) % 2 > 0;
 3500ms => (input / bit) % 2 > 0; 4000ms => (input / bit) % 2 > 0;
 4500ms => (input / bit) % 2 > 0;
 5s * @n + 10s => false;     5s * @n + 6s => (input / bit) % 2 > 0;
 5s * @n + 6500ms => (input / bit) % 2 > 0;
 5s * @n + 7000ms => (input / bit) % 2 > 0;
 5s * @n + 7500ms => (input / bit) % 2 > 0;
 5s * @n + 8000ms => (input / bit) % 2 > 0;
 5s * @n + 8500ms => (input / bit) % 2 > 0;
 5s * @n + 9000ms => (input / bit) % 2 > 0;
 5s * @n + 9500ms => (input / bit) % 2 > 0;
}
```

Listing 3. The serialized program in Listing 2

6 RELATED WORKS

One feature of TEFRP is that it exhibits different behavior for each time event for a single time-varying value. Event-driven FRP (E-FRP) [16], which this research refers to, describes multiple update expressions for each time-varying value. Events in E-FRP are based on interrupts, and each event does not occur simultaneously. Since TEFRP is based on time as the basis of events, each time description may refer to the same time. This differs from E-FRP in that it uses the first-match policy in terms of semantics, and discusses flexibility through logical combination and feasibility through serialization in terms of descriptiveness.

Yampa [5] is one of the arrowised FRP libraries. The switch combinator used in this library switches the expression to be updated by an event. Similar combinators are also implemented in Hailstorm [12], an arrow-based FRP language. These also realize a different update process for each event for time-varying values. On the other hand, the overhead of pattern matching for events is not small.

From a different perspective, Watanabe's context-oriented programming [6] extension of Emfrp [17] provides the ability to change the update method, called layers. In addition, an extension for automata to synchronous dataflow language Lustre [1] allows switching the method of updating variables in each state of the automaton. Since these functions do not switch for a single time-varying value, it is difficult to follow the behavior of a single time-varying value. On the other hand, TEFRP switching is forced on a single time-varying value. While this makes it easier to track a single time-varying value, it also causes the problem that the description is distributed when multiple time-varying values switch their behavior simultaneously for the same time event.

One FRP library that describes cycles over real-time is Hae [14]. This library directly describes the update period of time-varying values, so there is no need to fire time events externally as in TEFRP. The generated runtime defines the behavior based on the cycle. TEFRP has similar features but allows for more flexible descriptions, such as the ability to combine cycles with time descriptions.

EvEmfrp [15] is an FRP language that describes the update timing of time-varying values as cycles or interrupts. The update timing in EvEmfrp is described only for input and output time-varying values, while other time-varying values are inferred based on dependencies from inputs and outputs. This is in contrast to TEFRP, which requires that all time-varying values be given update timings explicitly. On the other hand, TEFRP allows multiple update expressions for a time-varying value, whereas in EvEmfrp, there is one update expression for a time-varying value. As a result, EvEmfrp has a syntax that controls the update timing to be inferred, such as a notation for referring to time-varying values that are updated at different times. Thus, there is a tradeoff between the amount of description and complication in TEFRP and EvEmfrp. Although there is an interrupt in the update timing in EvEmfrp, it is considered possible to implement a similar mechanism in TEFRP. Implementation of the mechanism is a future issue.

One language that uses real-time for embedded systems is, for example, Timed C [7]. Timed C can describe not only soft real-time but also hard real-time systems. In this respect, it is a language with a strong focus on scheduling. However, its scheduling is not based on state dependencies. In this respect, to implement two tasks that are independent of each other but share a state, it is necessary to carefully describe the temporal transitions of the tasks to avoid data races. On the other hand, however, it is possible to be very flexible in describing the scheduling. For example, limited to soft real-time, users can themselves describe the switching between skipping the next cycle by receiving the time beyond the deadline, performing the cycle from there, reducing the waiting time for the next one, and so on. We would like to consider extending the time description to express these functions in real-time systems in TEFRP.

7 CONCLUSIONS

We proposed TEFRP, a language that allows defining time-varying values with different update processing for time events. TEFRP is based on the assumption of soft real-time embedded systems, and update processing is performed using the elapsed time from the start of the system as the event. The description of elapsed time can be made by combining multiple patterns. In this paper, we showed that the description can be converted into a simple pattern. This property allows the next time event to be executed to be determined with little or no time calculation at runtime.

The TEFRP in this paper assumes a generous task density. It is assumed that all time-varying value update processes are completed before the time when another time-varying value update is next performed. Although this paper does not analyze this assumption, it is necessary to analyze the worst-case execution time of the update process at each time to guarantee this assumption. Since each update process is relatively simple, it is relatively easy to introduce an analysis of the worst-case execution time. However, since the number of clocks required for each step of the computation, and even the frequency, differs depending on the environment, (approximate) modeling that provides a correspondence with real-time will be a difficult part of the research.

Also, TEFRP does not consider asynchronous events at all. Introducing the event itself is relatively straightforward since it is enough to introduce an interruption as a time event. For events that occur in the middle of the update process, the update process for the event can be postponed or the update process can be reset, as in P-FRP [18]. In either case, there is no major difference from the system introduced in this paper, except that interrupts are assumed. On the other hand, asynchronous events may change the time of periodic tasks. Considering a system in which the state is initialized by a button, the cycle is readjusted according to the timing of the button press. To support such a system, we would consider introducing a description of time events based on the occurrence of asynchronous events in the TEFRP.

There are multiple possibilities for enhancing the expressive capabilities of TEFRP. Similar to Emfrp, the output node of a module could be used as a node definition. However, rather than simply reusing definitions, we would like to be able to change phases and cycles of update processes in the module. It is possible to make the time description flexible by allowing constant parameters to be defined in the module. However, the constraints explained in Section 4.3 cannot be described as linear integer constraints. It will be necessary to discuss how to handle them. In addition, since the definition of the node array [10] uses indices as constants, we can easily convert a time series data sequence (serial data) into an array data sequence (parallel data) by using this for time description. We plan to introduce this method in parallel.

Acknowledgements

This work was supported in part by JSPS KAKENHI Grant Numbers 19K20245, 21K11822 and 22K11967.

References

1. Colaço, J.L., Pagano, B., Pouzet, M.: A conservative extension of synchronous dataflow with state machines. In: Proceedings of the 5th ACM International Conference on Embedded Software. pp. 173–182. EMSOFT '05, ACM, New York, NY, USA (2005), https://doi.org/10.1145/1086228.1086261

2. Czaplicki, E., Chong, S.: Asynchronous functional reactive programming for GUIs. In: 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013). pp. 411–422. ACM (2013)
3. Elliott, C., Hudak, P.: Functional reactive animation. In: 2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997). pp. 263–273. ACM (1997)
4. Helbling, C.: Juniper language documentation (ver. 2.2.0). http://www.juniper-lang.org/language_docs.html (Nov 2016), http://www.juniper-lang.org/index.html
5. Hudak, P., Courtney, A., Nilsson, H., Peterson, J.: Arrows, robots, and functional reactive programming. In: Advanced Functional Programming, Lecture Notes in Computer Science, vol. 2638, pp. 159–187. Springer-Verlag (2003)
6. von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented programming: Beyond layers. In: Proceedings of the 2007 International Conference on Dynamic Languages: In Conjunction with the 15th International Smalltalk Joint Conference 2007. pp. 143–156. ICDL '07, ACM, New York, NY, USA (2007), https://doi.org/10.1145/1352678.1352688
7. Natarajan, S., Broman, D.: Timed c: An extension to the c programming language for real-time systems. In: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 227–239. IEEE (2018)
8. Pembeci, I., Nilsson, H., Hager, G.: Functional reactive robotics: An exercise in principled integration of domain-specific languages. In: 4th International Conreference on Principles and Practice of Declarative Programming (PPDP 2002). pp. 168–179. ACM (2002)
9. Peterson, J., Hudak, P., Elliott, C.: Lambda in Motion: Controlling Robots with Haskell. In: Proceedings of the First International Workshop on Practical Aspects of Declarative Languages. pp. 91–105. Springer-Verlag, Berlin, Heidelberg (1999)
10. Sakurai, Y., Moriguchi, S., Watanabe, T.: Functional reactive programming for embedded systems with GPGPUs. In: 10th International Conference on Software and Computer Applications (ICSCA '21). pp. 75–80. ACM (Feb 2021)
11. Salvaneschi, G., Hintz, G., Mezini, M.: REScala: Bridging between object-oriented and functional style in reactive applications. In: 13th International Conference on Modularity (Modularity 2014). pp. 25–36. ACM (2014)
12. Sarkar, A., Sheeran, M.: Hailstorm: A statically-typed, purely functional language for IoT applications. In: Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming. PPDP '20, ACM (2020), https://doi.org/10.1145/3414080.3414092
13. Sawada, K., Watanabe, T.: Emfrp: A functional reactive programming language for small-scale embedded systems. In: MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity. pp. 36–44. ACM (Mar 2016)
14. Sheng, W., Watanabe, T.: Functional reactive EDSL with asynchronous execution for resource-constrained embedded systems. In: Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing. Studies in Computational Intelligence, vol. 850, pp. 171–190. Springer (Aug 2019)
15. Sogo, K., Tsuji, Y., Moriguchi, S., Watanabe, T.: Periodic and aperiodic task description mechanisms in an frp language for small-scale embedded systems. In: Proceedings of the 10th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems. pp. 43–53. REBLS 2023, ACM, New York, NY, USA (2023), https://doi.org/10.1145/3623506.3623578

16. Wan, Z., Taha, W., Hudak, P.: Event-driven FRP. In: Practical Aspects of Declarative Langauges. Lecture Notes in Computer Science, vol. 2257, pp. 155–172. Springer-Verlag (2002)
17. Watanabe, T.: 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). pp. 23–30. ACM (Jul 2018)
18. Zou, X., Cheng, A.M., Jiang, Y.: P-FRP task scheduling: A survey. In: 2016 1st CPSWeek Workshop on Declarative Cyber-Physical Systems (DCPS). pp. 1–8. IEEE (2016)