Vol.2016-EMB-43 No.6
2016/11/11

Towards an Actor-Based Execution Model of an FRP Language for Small-Scale Embedded Systems

Takuo Watanabe[1,a)] Kensuke Sawada[1]

Abstract: Functional reactive programming (FRP) is a programming paradigm for reactive systems based on declarative abstractions to express time-varying values. Previous works showed that FRP is beneficial to embedded systems. In this paper, we propose a new execution mechanism for an FRP language designed for resource constrained embedded systems. The mechanism is based on the Actor model, a concurrent computation model in which computation is achieved by actors communicating via asynchronous messages. We adopt actors for the run-time representation of time-varying values and event streams. With this representation, we can naturally integrate asynchronous execution mechanism in the runtime of the language.

Keywords: Functional Reactive Programming, Embedded Systems, Actor Model

1. Introduction

Reactive systems are computational systems that respond to external events. Embedded systems are typical instances of reactive systems, in which changes in sensor values and switch states are examples of external events.

The order of events in a reactive system is usually not predictable, as they arrive asynchronously. Thus, describing reactive behaviors in conventional sequential programming languages is not straightforward. In practice, *polling* and *callbacks* are commonly used techniques to handle asynchronous events. However, they usually split the control flow of a program into multiple small pieces and thus are obstacles to modularity.

Functional Reactive Programming (FRP)[2] is a programming paradigm for reactive systems based on the functional (declarative) abstractions of time-varying values and events. Such abstractions are essential in FRP because we often employ continuously changing data over time as the sources of external events. Environmental sensor values are examples of such data. Time-varying values provide straightforward ways to express reactive behaviors. We can, of course, use them to represent discrete events.

FRP has been actively studied and recognized to be promising for various kinds of reactive systems including robots[4]. The application to robots suggests that FRP can be useful for other embedded systems. However, with a few exceptions, the majority of the FRP systems developed so far are Haskell-based, and therefore they require substantial runtime resources. Hence, it is virtually impossible to run such FRP systems on resource constrained platforms.

We designed and developed a new FRP language Emfrp that mainly targets small-scale embedded systems[6]. The term small-scale here means that the target platforms are not powerful enough to run conventional operating systems such as Linux. In contrast to other FRP languages, Emfrp does not treat time-varying values as first-class to guarantees that the amount of the runtime memory used by an Emfrp program is predictable.

The runtime system of Emfrp is based on a push-based, synchronous evaluation of time-varying values. However, we sometimes need to realize asynchrony for efficient execution[3]. Since Emfrp is a simple language specialized for the description of reactive behaviors, interfaces to external devices (including the runtime system) rely on libraries (I/O code) written in C. One problem caused by this design is that if we wish to add an asynchronous execution mechanism to the runtime, it might be realized as an ad-hoc C code.

To address the issue, we propose an integration of the Actor model[1] in Emfrp runtime, which provides a high-level view of the internals of the I/O code as well as a high-level abstraction for inter-device communication. In this integration, actors provide not only the representation of time-varying values, but also an asynchronous interface to the internals of the runtime.

The rest of the paper is organized as follows. The next section briefly describes Emfrp using an example of a simple air-conditioner controller. In Section 3, we present our Actor-based execution model. The section also shows the implementation of delayed nodes as an application to an asynchronous computation. Then Section 4 concludes the paper.

[1] Tokyo Institute of Technology
[a)] takuo@acm.org

```
1  module ACController   # module name
2  in   tmp : Float,     # temperature sensor
3       hmd : Float      # humidity sensor
4  out ac   : Bool       # air-conditioner
5  use Std               # standard library
6
7  # discomfort (temperature-humidity) index
8  node di = 0.81 * tmp + 0.01 * hmd
9            * (0.99 * tmp - 14.3) + 46.3
10
11 # air-conditioner switch
12 node ac = di >= 75.0
```

Fig. 1 Emfrp Module for an Air-Conditioner Controller

2. Overview of Emfrp

Emfrp[6][*1] is a purely functional programming language designed for resource constrained embedded systems. This section briefly describes the language with some examples.

2.1 Design Considerations

Designing abstraction mechanisms for time-varying values and events is the central topic of FRP language design. Most of existing FRP languages and libraries, such as Elm[3] or Yampa[4], treat time-varying values as first-class data that encapsulate time dependencies. Data types (or type constructors) for the purpose are either built-in (*e.g.*, `Signal` in Elm) or user-definable using type constructors such as arrows[5].

We adopt a different approach for Emfrp. We often represent a program in (functional) reactive style as a directed graph whose nodes and edges represent time-varying values and their dependencies respectively. The design of Emfrp directly reflects this representation. An Emfrp program consists of a fixed number of named *nodes* that express time-varying values. A node corresponds to a signal or a behavior in other languages.

Because Emfrp is mainly targeted at small-scale embedded systems, we designed the language to have the following characteristics to make the amount of runtime memory consumption predictable.

- Nodes (time-varying values) are not first-class value. We must, therefore, always specify nodes with their names.
- The language does not provide ways to alter the dependency relation between nodes at runtime. In other words, the graph representation of a program is static.
- Recursion is not allowed in function and type definitions.

2.2 Example: Air-Conditioner Controller

An Emfrp program consists of one or more *modules*. Fig. 1 is an example Emfrp module for a simple air-conditioner controller. It reads data from two environmental sensors (temperature and humidity) and turns an air-conditioner on only during the discomfort index[*2] calculated from the sensor values is more than or equal to 75, otherwise turns it off.

```
11 # air-conditioner switch
12 node init[False] ac = di >= 75.0 + ho
13
14 # hysteresis offset
15 node ho = if ac@last then -0.5 else 0.5
```

Fig. 2 Improved Air-Conditioner Controller

A module definition contains a single module header followed by one or more type, function or node definitions used in the module. In Fig. 1, the module header (lines 1–5) defines the module name (`ACController`), then declares two input nodes (`tmp` and `hmd`) and one output node (`ac`), and specifies the library module (`Std`) used in this module.

The rest of the module (lines 7–12) consists of two node definitions. A node definition looks like

node [**init**[c]] $n = e$

where n is the node name and e is an expression that describes the (time-varying) value of the node. The optional **init**[c] specifies the constant c as the initial value of the node. Note that if e contains another node name m, we say that n depends on m. While the value of m changes over time, the value of n varies also.

Emfrp has three kinds of nodes: *input*, *output* and *internal*. Each input or output node has a connection to an external device, while an internal node has no such connection. The value of an input node always expresses the current value (or state) of the device connected, and the value of an output node acts on its device. Thus, an input node needs no node definition in the module. In contrast, other kinds of nodes require explicit definitions to determine their values.

In the example, `tmp` and `hmd` are input nodes connected to the sensors. Their values represent the current environmental data. The internal node `di` (lines 8–9) always expresses the latest discomfort index depending on `tmp` and `hmd`. The output node `ac` (line 12) serves as a time-varying Boolean value that controls the on/off status of the air-conditioner.

2.3 Expressing History-Sensitive Behaviors

In fact, the air-conditioner controller in Fig. 1 has a serious flaw. Let us consider a situation that the discomfort index drifts around the threshold (75.0). In such a case, the time-varying Boolean value of the output node `ac` may change at a fast rate, which results in quick changes of the on/off status of that are hazardous to the air-conditioner

To avoid such situation, we add a history-sensitive behavior (hysteresis) to the controller by replacing lines 11–12 in Fig. 1 with Fig. 2. This patch adds a new internal node `ho` that represents a history-sensitive offset to the threshold. The node definition of `ho` has an expression `ac@last`, which

[*1] https://github.com/sawaken/emfrp/

[*2] a.k.a. temperature-humidity index. About 50% of people feel uncomfortable if it reaches 75.

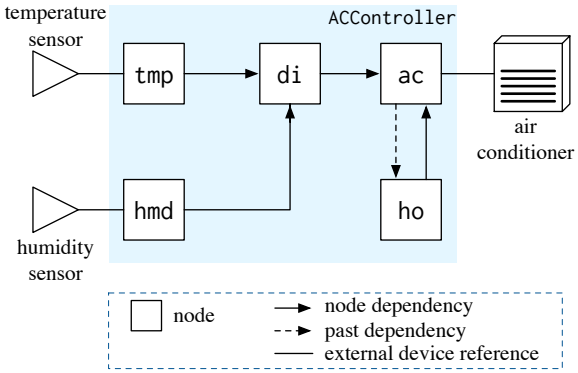

Fig. 3 Graph Representation of Fig. 2

refers to the value of `ac` at the "previous moment" — the value evaluated in the previous *iteration* (see Section 2.4).

The new program behaves as follows. While the air-conditioner is off, namely, `ac` is `False`, `di` must be more than or equal to 75.5 to turn it `True`. Once it becomes `True`, `di` must be less than 74.5 to turn it `False`. As a result, we can avoid the quick changes of the on/off status explained above.

The operator `@last` in Emfrp generalizes `foldp` in Elm. The latter only allows a node to refer to the previous value of itself, whereas the former provides access to those of arbitrary nodes. Owing to this simple operator and other features, Emfrp offers a flexible and intuitive way of describing reactive behaviors.

2.4 Execution Model

As described in Section 2.1, an Emfrp program can be represented as a directed graph whose nodes and edges correspond to time-varying values and their dependencies respectively. Figure 3 shows the graph representation of Fig. 2, which consists of five nodes and five edges.

We categorize the edges (dependencies) into two kinds: *past* and *present*. A past edge from node m to n means that n has m`@last` in its definition. A present edge from node m to n, in contrast, means that n directly refers to m. In Figure 3, the dotted arrow line from `ac` to `ho` is the past edge. All other edges are present.

By removing the past edges from the graph representation of an arbitrary Emfrp program, we obtain a directed-acyclic graph (DAG). The topological sorting on the DAG gives a sequence of the nodes. For Figure 3, we have: `tmp`, `hmd`, `ho`, `di`, `ac`.

The execusion model of Emfrp is push-based[2]. The runtime system updates the values of the nodes by repeatedly evaluating the elements of the sequence. We call a single evaluation cycle an *iteration*. The order of updates (scheduling) in an iteration must obey the partial order determined by the above mentioned DAG.

The value of n`@last` is the value of n in the last iteration. At the first iteration, where no nodes have their previous values, n`@last` refers to the initial value c specified with **`init`**`[`c`]` in the definition of n. The definition of `ac` in the modified program (Fig. 2) has the initial value as `ho` refers to `ac@last`.

```
class Actor {
public:
    virtual void send(Message *m);
    virtual void receive(Message *m);
    virtual void activate(Message *m) = 0;
}
```

Fig. 4 C++ Class for Actors

```
class TMPNode : public Actor {
public:
    TMPNode(Actor2 *di, TMPSensor *tmp);
    virtual ~TMPNode() {}
    virtual void activate(Message *m);
private:
    Actor2 *di;
    TMPSensor *tmp;
}

void TMPNode::activate(Message *m) {
    di->send1(
        Message::floatMessage(tmp->read(),
                              m->cust));
}

class DINode : public Actor2 {
public:
    DINode(Actor *ac) ac(ac) { ... }
    virtual ~DINode() {}
    virtual void activate(Message *m);
private:
    Actor *ac;
}

void DINode::activate(Message *mt,
                      Message *mh) {
    assert(mt->cust == mh->cust);
    float t = mt->getFloat();
    float h = mh->getFloat();
    float di = 0.81 * t + 0.01 * h
               * (0.99 * t - 14.3) + 46.3;
    ac->send(
        Message::floatMessage(di, mt->cust));
}
```

Fig. 5 Actors for Nodes `tmp` and `di`

3. Integration of the Actor Model

This section briefly describes an integration of the Actor model in Emfrp. In this integration, each node is represented by an actor and a dependency between two nodes is expressed as an actor reference. As a natural consequence, iterations are realized by message passing. The actor-based representation provides a higher-level abstraction for nodes in the I/O code of an Emfrp program.

3.1 Representing Nodes as Actors

We use C++ objects to represent actors. The class `Actor` (Fig. 4) provides the basic actor APIs. The method `send` puts a message in the system queue. When the message is scheduled to be received by an actor, `receive` and `activate` are invoked at the receiver in this order.

```
1  module ACController  # module name
2  in  tmp : Float,     # temperature sensor
3      hmd : Float      # humidity sensor
4      pulse10ms : Bool # 10 msec interval timer
5  out ac  : Bool,      # air-conditioner
6      led : Bool       # LED
7  use Std              # standard library
8
9  # discomfort (temperature-humidity) index
10 node di = 0.81 * tmp + 0.01 * hmd
11           * (0.99 * tmp - 14.3) + 46.3
12
13 node init[0] timer =
14      if !pulse10ms@last && pulse10ms
15      then (timer@last + 1) % 600
16
17 # air-conditioner switch
18 node ac = if timer@last != timer && timer == 0
19           then di >= 75.0
20
21 # LED blinks at 1Hz
22 node led = (timer % 100) < 50;
```

Fig. 6 Air-Conditioner Controller using a Timer

The compiler for the actor-integrated version of Emfrp is supposed to produce a collection of actor classes that represent the nodes in the original Emfrp program. Fig. 5 shows the definitions of actors that represent nodes `tmp` and `di`. The class `Actor2` is a *join actor*[*3] that requires two messages to invoke `activate`. Join actors represent nodes that depend on multiple nodes. We have `Actor3`, `Actor4`, ... as well.

The compiler also generates a piece of code that instantiates actors in static area as follows.

```
ACNode ac();
HONode ho(&ac);
DINode di(&ac);
TMPNode tmp(&di);
HMDNode hmd(&di);
```

Since node dependencies are static in Emfrp, actor references are provided as arguments of constructors. A single iteration starts with messages to actors that represent the input nodes as follows.

```
tmp->send(Message::unitMessage(&sys_actor));
hmd->send(Message::unitMessage(&sys_actor));
```

The iteration ends with messages to the actor `sys_actor` sent from the actors that represent the output nodes.

3.2 Example: Air-Conditioner Controller using a Timer

Timers are crucial components of most embedded systems. Fig. 6 shows another implementation of the air-conditioner controller that utilizes a timer. In this implementation, the changes of the on/off status occur at most once per minute. The input node `pulse10ms` is connected to a hardware interval timer with 10 msec interval. The internal node `timer` constantly counts up on each rising edge of `pulse10ms` and resets to 0 every one minute. The value of `ac` may change only when `timer` becomes 0 and `di` ≥ 75. In addition, an LED blinks at 1Hz to indicate that the system is in operation.

A possible problem of this code is that, due to the push-based execution model of Emfrp, `di`, `tmp` and `hmd` are updated in every iteration regardless of their necessities. In fact, however, we can see from the definition of `ac` (lines 18–19) that the value of `di` (hence `tmp` and `hmd`) is required only once per minute. The results of all other updates are just ignored. Such wasteful computation is unfavorable especially for small-scale embedded systems since it leads to higher power consumption.

[*3] Similar notion to *join continuation*[1]

```
17 # air-conditioner switch
18 node ac = if timer@last != timer && timer == 0
19           then ( di >= 75.0 )@delay
```

Fig. 7 Delayed Block

```
1  class DINode : public Actor2 { ... }
2
3  void DINode::activate(Message *mt,
4                        Message *mh) {
5      float t = mt->getFloat();
6      float h = mh->getFloat();
7      float di = 0.81 * t + 0.01 * h
8                 * (0.99 * t - 14.3) + 46.3;
9      mt->cust->send(
10         mkFloatMessage(di, mt->cust));
11 }
12
13 class ACNode : public Actor { ... }
14
15 void ACNode::activate(Message *m) {
16     if (m->prevInt() != m->getInt() &&
17         m->getInt() == 0) {
18         tmp->send(Message::unitMessage(
19                        &acDelayedBlock));
20         hmd->send(Message::unitMessage(
21                        &acDelayedBlock));
22     }
23 }
24
25 class ACDelayedBlock : public Actor { ...}
26
27 void ACDelayedBlock::activate(Message *m) {
28     m->cust->send(
29         Message::booleanMessage(
30             m->getFloat() > 75.0, m->cust));
31 }
```

Fig. 8 Implementation of the Delayed Block

3.3 Delayed Blocks

The problem described in the previous subsection can be resolved using a pull-based execution model. However, the execution of periodically updating nodes such as `timer` and `LED` require push-based model. Thus we need a mixture of both execution models.

We extend Emfrp with a simple mechanism called *delayed block*. Syntactically, a delayed block is an expression suffixed with `@delay`. Fig. 7 shows an example use of a delayed block with which lines 17–19 of Fig. 6 should be replaced.

In Fig. 6, node `ac` depends on both `timer` and `di`. How-

ever, in Fig. 7, `di` is removed from the dependency of `ac`. So `di` is no longer in the dependency of any output nodes. This means that `di`, hence `tmp` and `hmd` are removed from the program graph. The value of `di` is needed only when the condition of if statement of Fig. 7 holds. Thus, the compiler performs a simple dependency analysis and produces the code so that starting messages to `tmp` and `hmd` are sent when the condition holds (Lines 6–9 in Fig. 8).

The compiler now treats `DINode` as an output node. Thus the result will be passed to the actor `acDelayedBlock`, which plays a role of the continuation of the starting messages to `tmp` and `hmd`. As a result, the sensor values and the discomfort index value are calculated only if the condition regarding the timer is satisfied.

4. Concluding Remark

This paper briefly describes a simple idea of integrating the Actor model into Emfrp, a functional reactive programming language designed for resource constrained embedded systems. The integration provides a higher-level view of the internal representation of nodes, representations of time-varying values, as well as an actor-based inter-device communication mechanism.

The group of actors representing the nodes of an Emfrp program are viewed as the meta-level of the program. Thus it is possible to apply a variation of group-wide reflection[7] to the actor group to realize more drastic customization such as application-oriented evaluation (scheduling) polities or dynamic node reconfiguration.

We have just started this project. We need to work on the abstraction of the APIs and the integration of inter-device communication mechanism.

Acknowledgments

This work is supported in part by JSPS KAKENHI Grant No. 15K00089.

References

[1] Agha, G.: *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press (1986).
[2] Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S. and De Meuter, W.: A Survey on Reactive Programming, *ACM Computing Surveys*, Vol. 45, No. 4, p. 52 (online), DOI: 10.1145/2501654.2501666 (2013).
[3] Czaplicki, E. and Chong, S.: Asynchronous Functional Reactive Programming for GUIs, *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, ACM, pp. 411–422 (online), DOI: 10.1145/2499370.2462161 (2013).
[4] Hudak, P., Courtney, A., Nilsson, H. and Peterson, J.: Arrows, Robots, and Functional Reactive Programming, *Advanced Functional Programming*, Lecture Notes in Computer Science, Vol. 2638, Springer-Verlag, pp. 159–187 (online), DOI: 10.1007/978-3-540-44833-4˙6 (2003).
[5] Hughes, J.: Generalising monads to arrows, *Science of Computer Programming*, Vol. 37, No. 1–3, pp. 67–111 (online), DOI: 10.1016/S0167-6423(99)00023-4 (2000).
[6] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *Modularity 2016 Constrained and Reactive Objects Workshop (CROW 2016)*, ACM, pp. 36–44 (online), DOI: 10.1145/2892664.2892670 (2016).
[7] Watanabe, T.: Towards a Compositional Reflective Architecture for Actor-Based Systems, *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2013)*, ACM, pp. 19–24 (online), DOI: 10.1145/2541329.2541341 (2013).