Towards an Integration of the Actor Model in an FRP Language for Small-Scale Embedded Systems

Takuo Watanabe   Kensuke Sawada

Department of Computer Science, Tokyo Institute of Technology

takuo@acm.org, sawada@psg.c.titech.ac.jp

Abstract

This paper presents an integration of the Actor model in Emfrp, a functional reactive programming language designed for resource constrained embedded systems. In this integration, actors not only express nodes that represent time-varying values, but also present communication mechanism. 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.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Control structures; D.3.2 [*Language Classifications*]: Applicative (functional) languages

General Terms Languages, Design

Keywords Functional Reactive Programming, Embedded Systems

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)[3–5] 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[5]. 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[7]. 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.

Since Emfrp is a simple language specialized for the description of reactive behaviors, interfaces to external devices rely on libraries (I/O code) written in C. One problem caused by this design is that if we wish to add an inter-device communication mechanism, 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 inter-device communication mechanism but also the representation of time-varying values.

2. Overview of Emfrp

Emfrp[7][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[5], 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[6].

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 character-

[Copyright notice will appear here once 'preprint' option is removed.]

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

```
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
```

Listing 1. Emfrp Module for an Air-Conditioner Controller

istics 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*. Listing 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.

A module definition contains a single module header followed by one or more type, function or node definitions used in the module. In Listing 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

$$\texttt{node}\ [\texttt{init}[c]]\ n = e$$

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

```
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
```

Listing 2. Improved Air-Conditioner Controller

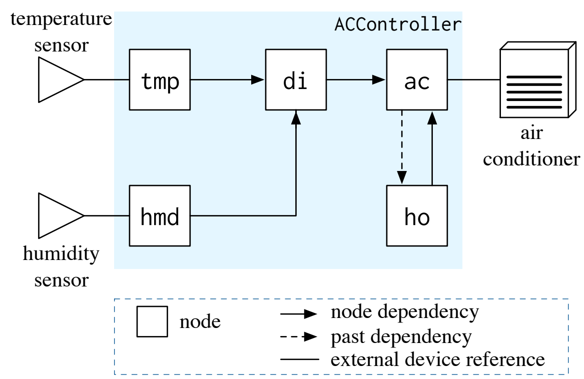

Figure 1. Graph Representation of Listing 2

2.3 Expressing History-Sensitive Behaviors

In fact, the air-conditioner controller in Listing 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 Listing 1 with Listing 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 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 1 shows the graph representation of Listing 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 1, 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

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

Listing 3. C++ Class for Actors

(DAG). The topological sorting on the DAG gives a sequence of the nodes. For Figure 1, 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 (Listing 2) has the initial value as `ho` refers to `ac@last`.

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` (Listing 3) 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.

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. Listing 4 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. Listing 5 shows another implementation of the air-conditioner con-

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

```
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(), 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));
}
```

Listing 4. Actors for Nodes `tmp` and `di`

```
module ACController   # module name
in  tmp : Float,      # temperature sensor
    hmd : Float       # humidity sensor
    pulse10ms : Bool # 10 msec interval timer
out ac  : Bool,       # air-conditioner
    led : Bool        # LED
use Std               # standard library

# discomfort (temperature-humidity) index
node di = 0.81 * tmp + 0.01 * hmd
          * (0.99 * tmp - 14.3) + 46.3

node init[0] timer =
     if !pulse10ms@last && pulse10ms
     then (timer@last + 1) % 600

# air-conditioner switch
node ac = if timer@last != timer && timer == 0
          then di >= 75.0

# LED blinks at 1Hz
node led = (timer % 100) < 50;
```

Listing 5. Air-Conditioner Controller using a Timer

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

Listing 6. Delayed Block

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

Listing 7. Implementation of the Delayed Block

troller 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.

3.3 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`. Listing 6 shows an example use of a delayed block with which lines 17–19 of Listing 5 should be replaced.

In Listing 5, node `ac` depends on both `timer` and `di`. However, in Listing 6, `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 Listing 6 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 Listing 7).

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 short 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[8] 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.

References

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

[2] E. Bainomugisha, A. L. Carreton, T. Van Cutsem, S. Mostinckx, and W. De Meuter. A survey on reactive programming. *ACM Computing Surveys*, 45(4):52, 2013. doi: 10.1145/2501654.2501666.

[3] E. Czaplicki and S. Chong. Asynchronous functional reactive programming for GUIs. In *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, pages 411–422. ACM, 2013. doi: 10.1145/2499370.2462161.

[4] C. Elliott and P. Hudak. Functional reactive animation. In *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*, pages 263–273. ACM, 1997. doi: 10.1145/258949.258973.

[5] P. Hudak, A. Courtney, H. Nilsson, and J. Peterson. Arrows, robots, and functional reactive programming. In *Advanced Functional Programming*, volume 2638 of *Lecture Notes in Computer Science*, pages 159–187. Springer-Verlag, 2003. doi: 10.1007/978-3-540-44833-4_6.

[6] J. Hughes. Generalising monads to arrows. *Science of Computer Programming*, 37(1–3):67–111, 2000. doi: 10.1016/S0167-6423(99)00023-4.

[7] K. Sawada and T. Watanabe. Emfrp: A functional reactive programming language for small-scale embedded systems. In *Modularity 2016 Constrained and Reactive Objects Workshop (CROW 2016)*, pages 36–44. ACM, Mar. 2016. doi: 10.1145/2892664.2892670.

[8] T. Watanabe. Towards a compositional reflective architecture for actor-based systems. In *Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2013)*, pages 19–24. ACM, Oct. 2013. doi: 10.1145/2541329.2541341.