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)\cite{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\cite{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\cite{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\cite{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\cite{7} 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\cite{3} or Yampa\cite{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\cite{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-

\footnote{https://github.com/sawaken/emfrp/}

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

\[
\text{node [init]}[c] \ n = e
\]

where \( n \) is the node name and \( e \) is an expression that describes the (time-varying) value of the node. The optional \( \text{init}[c] \) 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: \textit{input}, \textit{output} and \textit{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 \( \text{tmp} \) and \( \text{hmd} \). The output node \( ac \) (line 12) serves as a time-varying Boolean value that controls the on/off status of the air-conditioner.

\[
di = 0.81 \times \text{tmp} + 0.01 \times \text{hmd}
\]

\[
ac = \text{di} \geq 75.0
\]

\[
ode init[False] ac = di \geq 75.0 + ho
\]

\[
ode ho = \text{if} \ ac@\text{last} \text{ then } -0.5 \text{ else } 0.5
\]

Listing 2. Improved Air-Conditioner Controller
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 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.

```cpp
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.

```cpp
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-

```cpp
Listing 3. C++ Class for Actors
```
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 \( di \geq 75 \). 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 periodical updating nodes such as \( timer \) and \( LED \) require push-based model. Thus we need a mixture of both execution models. 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


---

### Listing 6. Delayed Block

```java
class DINode : public Actor { ...

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

class ACDelayedBlock : public Actor { ...

void ACDelayedBlock::activate(Message *m) {
    if (t > prevInt() != m->getInt() &&
        m->getInt() == 0) {
        //acDelayedBlock
    }
}
```

### Listing 7. Implementation of the Delayed Block

```java
class DINode : public Actor { ...

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

class ACDelayedBlock : public Actor { ...

void ACDelayedBlock::activate(Message *m) {
    if (m->prevInt() != m->getInt() &&
        m->getInt() == 0) {
        //acDelayedBlock
    }
}
```

---

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.