Vol.2018-ARC-230 No.26
Vol.2018-SLDM-183 No.26
Vol.2018-EMB-47 No.26
2018/3/8

アクターモデルと関数リアクティブプログラミングの融合による小規模組込みシステム開発

渡部 卓雄[1,a)]

概要：アクターモデルの考え方を取り入れた関数リアクティブプログラミング (FRP) 言語の利用が，計算資源の限られた組込みシステムにおいて柔軟な実行方式を可能にすることを示す．提案方式の特色は，小規模組込みシステム向け FRP 言語の実行時システムを並行計算モデルのひとつであるアクターモデルを用いて構成することで，FRP の特徴である宣言的記述にもとづく高い抽象性を保ちつつ，適応性の向上を可能にすることにある．これにより，適応的動作を含む組込みシステムの動作を宣言的に記述することが可能になることを例題を通して示す．

キーワード：アクターモデル，関数リアクティブプログラミング, 組込みシステム

Developing Small-Scale Embedded Systems using an Integration of the Actor Model in Functional Reactive Programming

Takuo Watanabe[1,a)]

Abstract: This paper briefly describes 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 adaptable execution mechanism in the runtime of the language.

Keywords: Actor Model, Functional Readtive Programming, Embedded Systems

1. はじめに

離散的なイベントや連続的な外界の状態変化などの入力に対して，応答の生成および自身の状態の更新をし続けるシステムを**リアクティブシステム** *(reactive system)* と呼ぶ．組込みシステムや GUI などのインタラクティブなシステムはリアクティブシステムの典型例である．

一般にリアクティブシステムに対する入力は，与えられる順序およびタイミングはあらかじめ予測できないものとされる．このことに対処するため，リアクティブシステムのプログラミングでは，コールバック，イベントループ，ポーリング等が用いられる．これらは一般的な手続き型プログラミング言語においても実現可能である一方，コードの細分化を招き可読性低下の原因となる．

リアクティブプログラミング *(Reactive Programming)* は，**イベントストリーム**や**時変値** *(time-varying value)* と呼ばれる抽象化機構を用いて入力およびそれらに依存する値を表現することで，リアクティブシステムの効果的な記述を支援するプログラミングパラダイムである [2]．また**関数リアクティブプログラミング** *(Functional Reactive Programming, FRP)* は，関数プログラミングに上記の抽象化機構を導入することでリアクティブシステムの宣言的な記述を可能にする．

著者らは，標準的な C をターゲットとした純粋関数型言語である Emfrp を設計・実装し，いくつかの例題をとおし

[1] 東京工業大学
Tokyo Institute of Technology
[a)] takuo@acm.org

Vol.2018-ARC-230 No.26
Vol.2018-SLDM-183 No.26
Vol.2018-EMB-47 No.26
2018/3/8

```
1  module FanController  # module name
2  in  tmp : Float,      # temperature sensor
3      hmd : Float       # humidity sensor
4  out fan : Bool        # fan switch
5  use Std               # standard library
6
7  # discomfort (temperature–humidity) index
8  node di =
9      0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
10
11 # fan switch
12 node init[False] fan = di >= 75.0 + ho
13
14 # hysteresis offset
15 node ho = if fan@last then -0.5 else 0.5
```

図 1 温度・湿度センサによるファン制御プログラム

てその有用性を明らかにしている [4]．Emfrp では時変値を一級データとして扱わず，必ず名前によって参照するというプログラミング上の制約を設けている．また一般的な再帰呼び出しおよび再帰的なデータ構造の利用を禁止する代わりに，任意の時変値の直前値を参照できる言語機構を導入している．以上によってプログラムが利用する記憶領域の大きさを静的に確定できるようになっている．また実行系としてシンプルな push 型を採用しているため，生成されるコードサイズも小さい．

このように Emfrp ではプログラムの表現力を大幅に落とすことなく小規模組み込みシステムに適した言語機構を提供している．しかしその一方でプログラムの実行方式に関する柔軟性が不足しており，例えば pull 型の実行方式を採用すれば避けられるような不要な計算を行ってしまう場合がある．

このような問題は push 型と pull 型の実行を混在させることで解決できるが，そうすることで実行系が複雑になりサイズが肥大化するという問題が生じる．筆者らはアクターモデルを実行系に導入することで，この問題を解決可能であることを示した [5]．本稿ではその提案方式の概略を述べ，その実現方式と応用例を示し，アクターモデルの導入が動的・適応的な動作を容易に実現できることを例を通して示す．

2. Emfrp

Emfrp[4] はマイクロコントローラ等の小規模組み込みシステム向けに設計された関数リアクティブプログラミング言語である．本節では同言語の設計と実装の概略について例を通して説明する．

2.1 例題：温度・湿度センサによるファンの制御

例題として，温度・湿度センサによるファンの制御プログラムを図 1 に示す．このプログラムはセンサの測定値から不快指数を計算し，その値が規定値（ここでは 75）以上になったときにファンのスイッチを ON にする．ただし不快指数の値が規定値近辺を上下したときにスイッチの ON/OFF が頻繁に行われることを防ぐため，簡単なヒステリシス制御を行なっている．

2.2 モジュールとノード

Emfrp のプログラムはモジュールと呼ばれる単位で構成される．図 1 のプログラムは `FanController` という名前を持つ 1 個のモジュールからなる．モジュールのヘッダ部（例では 1–5 行目）ではモジュール名や入力・出力ノード（後述）および使用するライブラリが宣言され，本体（例では 6 行目以降）では定数，関数およびノード（後述）が定義される．

ノード *(node)* とは Emfrp における時変値を表す言語機構であり，入力ノード，出力ノード，内部ノードの 3 種類に分類される．入力・出力ノードはそれぞれ外部機器からの入力と外部機器への出力を表している．この例では `tmp` および `hmd` が入力ノード，`fan` が出力ノードであり，それぞれ温度と湿度の現在値およびファンの状態 (ON/OFF) を表している．入力・出力ノード以外のノードを内部ノードと呼ぶ．この例では `di` と `ho` が内部ノードであり，それぞれ不快指数の現在値とヒステリシス制御のためのオフセットを表している．

出力ノードおよび内部ノードはモジュールの本体において予約語 **`node`** を用いて定義される．例題では 8-9 行目，12 行目，15 行目がそれぞれ `di`, `fan`, `ho` を定義している．入力ノードの値は外部機器によって決まるので **`node`** による定義は行われない．

各ノードの現在値は=の右辺の式（ノードの定義式と呼ぶ）によって決まる．ノードの定義式中に他のノード名が出現するとき，前者は後者に依存するという．例えばノード `di` は `tmp` および `hmd` に依存している．

定義式中におけるノード名はノードの現在値を表しているが，15 行目の `fan@last` のようにノード名の後に`@last` を伴う場合は，当該ノードの**直前値** *(previous value)* を表している．これは現在値に至る直前にノードが取っていた値であり，これを用いることで状態に依存したプログラムを記述することができる．例題ではヒステリシス制御のためのオフセットがファンの状態に依存していることをこの機構を用いて表現している．

2.3 プログラムのグラフ表現と実行モデル

Emfrp のプログラムはノードを頂点，ノード間の依存関係を有向辺とするグラフとして表現できる．図 1 のプログ

Vol.2018-ARC-230 No.26
Vol.2018-SLDM-183 No.26
Vol.2018-EMB-47 No.26
2018/3/8

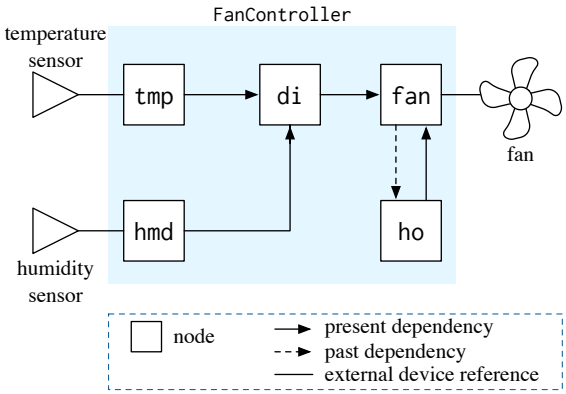

図 2 図 1 のグラフ表現

ラムをグラフとして表現したものを図 2 に示す．

ノード間の依存関係を表す辺のうち点線のものはノードの直前値による依存関係を表している（図 2 ではノード fan から ho への辺）．Emfrp では，プログラムを表すグラフから点線の辺を取り除いたものが有向非循環グラフ (DAG) となる必要がある．

Emfrp の実行系は，ノードの値を入力ノードから順番に更新してそれぞれの現在値を計算する．このときの順番は上で述べた DAG で表される半順序関係と矛盾しなければよい．つまり DAG をトポロジカルソートしたリストの順に更新を行なう．この例題の場合，例えば tmp, hmd, di, ho, fan の順で更新を行えばよい．ここでプログラムを構成するノード全ての更新を**サイクル**と呼ぶ．Emfrp の実行はサイクルの繰り返しとして実現できる．

各サイクルにおけるノードの直前値の参照は，当該ノードの更新前の値の参照として実現できる．この例では ho の現在値を計算するために fan が既に保持している値を参照すればよい．

2.4 組込みシステムで実行するための制約条件

Emfrp ではノードは必ず定義時に付与された名前で参照され，ノードそのものを変数に代入したり，関数の引数にすることはできない．つまりノードは一級データではない．また実行時におけるノードの生成・削除，および依存関係の変更はできない．以上より，プログラムを表すグラフはコンパイル時に決定され，実行時に変化することはないことが保証される．

またノードの値となりうる型は，真偽値，整数や浮動小数点数のような組込みの型，およびそれらを引数にもつ代数型に限定される．また Emfrp では再帰的な代数型は定義できない．これによりノードの値のサイズは静的に決定される．

加えて，Emfrp は繰り返し構文を持たず，かつ関数定義における再帰も禁止されている．したがって，ノードの定義式の計算は（組込み関数や演算子の計算が有限時間で停止する限り）有限時間で停止する．

以上の性質より，Emfrp ではプログラムとそれが必要とする記憶領域の大きさはコンパイル時に決定される．

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

図 3 C++ Class for Actors

3. アクターを用いた実行モデル

本節ではアクターを用いた Emfrp の実行手法について概要を述べる．提案手法では，各ノードはアクターとして，またノード間の依存関係はアクターの参照関係でそれぞれ表現される．そしてサイクルは DAG に沿ったメッセージの送受信によって実現される．

3.1 アクターによるノードの表現

ここではアクターを C++のオブジェクトとして実現する．図 3) に示すクラス Actor は基本的なアクターとしての振る舞いを提供する．メソッド send はメッセージをシステムキューに追加する．システムキュー内のメッセージは順にスケジュールされる．キューの先頭のメッセージはその宛先となるアクターのメソッド receive によって受信される．その後 activate によって当該メッセージに対応するメソッドの実行が行われる．

コンパイラは Emfrp モジュールのソースコードから，モジュール内のノードを表すアクターのクラスを生成する．図 4 はノード tmp と di を表すアクターの定義である．

クラス Actor2 は **合流アクター** *(join actor)*[*1] と呼ばれるものであり，指定された 2 つのメッセージが届いて初めてメソッド activateが実行される．合流アクターは 2 つのノードに依存するノードを表現している．同様に，3 つ以上のノードに依存するノードを表すために，Actor3, Actor4, ... などを用いる．

またコンパイラはアクターをスタティック領域に生成するために以下のようなコードを出力する．

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

Emfrp ではノード間の参照関係は静的に決定されるため，ノードを表すアクター同士の参照関係は上のようにコンス

[*1] **合流継続** *(join continuation)*[1] に類似した動作を行う

```
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));
}
```

図 4 Actors for Nodes tmp and di

トラクタの引数として与えられる．

また 1 回のサイクルは以下のように入力ノードを表すアクターへのメッセージとして開始される．

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

そして 1 回のサイクルの終了は，出力ノードを表すアクターから sys_actor というアクターへのメッセージとして表される．

3.2 例題：タイマを用いたファンの制御

タイマは組込みシステムにとって欠かせない要素である．

```
module FanController  # module name
in  tmp : Float,      # temperature sensor
    hmd : Float       # humidity sensor
    pulse10ms : Bool  # 10 msec interval timer
out fan : Bool,       # fan switch
    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 fan = if timer@last != timer && timer == 0
           then di >= 75.0

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

図 5 タイマを用いたファンの制御

図 5 はファン制御プログラムをタイマを用いて実装したものである．. この実装ではファンの ON/OFF は 1 分あたり高々 1 回だけ行うようにしている．入力ノード pulse10ms は 10ms 周期のハードウェアタイマに接続されているものとする．内部ノード timer は pulse10ms の立ち上がりエッジをカウントし，1 分ごとに 0 にリセットされる．ノード fan の値は timer が 0 にリセットされたときのみ変更され，それ以外のときは同じ値を保持する．加えて，動作中であることを示す LED が 1 秒周期で点滅する．

オリジナルの Emfrp の実行モデルでこのコードを実行すると各サイクル毎に全ノードの更新が行われる．しかしこのプログラムの場合，di およびそれが依存する tmp, hmd の値は timer の値が 0 になったとき（1 分に 1 回）しか必要とされず，それ以外の場合の値は無視される．アプリケーションの性質からも，各サイクル毎に温度と湿度を測定する必要はない．このような無駄な計算は特に小規模組込みシステムにおいては消費電力の増大につながるため避けるべきである．

3.3 遅延ブロック

前節で述べた問題は pull 式の実行によってある程度解決可能である．しかし本例題における timer や LED のような周期的な変化を伴うノードを持つプログラムには pull 式の実行は適切ではない．したがって pull 式と push 式の混在

Vol.2018-ARC-230 No.26
Vol.2018-SLDM-183 No.26
Vol.2018-EMB-47 No.26
2018/3/8

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

図 6 遅延ブロック

```
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 }
```

図 7 遅延ブロックの実装

した実行モデル [3] などが必要となるが，そのような実行モデルをリソースの限られた環境で実現するのは難しい．

ここでは **遅延ブロック** *(delayed block)* というアイデアを導入する．構文上は，遅延ブロックは@delay というサフィックスを持つ式として表現される．図 6 は遅延ブロックの例であり，図 5 の 17–19 行目をこの図で置き換えることを意図している．

図 5 では，ノード fan は timer と di の両方に依存している．しかし図 6 では，fan の di への依存は削除されている．したがって，di はどの出力ノードからも依存されなくなる．このことは，di そして tmp および hmd はプログラムを表すグラフから削除してよいことになる．

しかし di の値は図 6 における if 式の条件が成立したときのみは必要となる．そのためコンパイラは簡単な依存性解析を行い，当該条件が成立したときに tmp と hmd にメッセージを送信するようなコードを生成する（図 7 の 6–9 行目）．

ここでコンパイラは DINode を出力ノードのように扱う．そのため結果は上記で tmp と hmd に送信されたメッセージの継続に相当する acDelayedBlock に送信される．結果として，2 つのセンサの値と不快指数の値はタイマの値が 0 になったときのみ更新されることになり，無駄な計算を避けることができる．

4. まとめ

アクターモデルを小規模組込みシステム向け関数リアクティブプログラミング言語 Emfrp の実行系に統合する方式について概要を述べた．そして例を通して提案方式がプログラムの適応的な実行を宣言的に記述することに貢献することを示した．

本稿で例題に用いた @delayed のような（構文的な）仕組みを現在のところアドホックに実装しているが，FRP 向けの自己反映計算機構 [6] を用いた実現も可能であると考えられる．ただしその場合コンパイラが行う依存性解析も同時に導入する必要があるため，コンパイラを含めた自己反映計算のためのフレームワークの実現が必要となる．

また提案方式は，FRP にもとづいて実装されたプログラムにアクターモデルによる非同期通信の統合に用いることも可能である．現在，FRP を用いたセンサーネットワークの提案方式にもとづく実現について検討中である．

Acknowledgments

本研究の一部は JSPS 科研費 15K00089 の助成を受けている．

参考文献

[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] Elliott, C.: Push-Pull Functional Reactive Programming, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell*, pp. 25–36 (online), DOI: 10.1145/1596638.1596643 (2009).
[4] 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).
[5] Watanabe, T. and Sawada, K.: Towards an Integration of the Actor Model in an FRP Language for Small-Scale

Embedded Systems, *6th International Workshop on Programming based on Actors, Agents, and Decentralized Control (AGERE!@SPLASH 2016)* (2016).

[6] Watanabe, T. and Sawada, K.: Towards Reflection in an FRP Language for Small-Scale Embedded Systems, *Companion to the 1st International Conference on the Art, Science and Engineering of Programming (Programming 2017)*, ACM, pp. 10:1–10:6 (online), DOI: 10.1145/3079368.3079387 (2017).