小規模組込みシステム向けFRP言語への文脈指向プログラミング機構の導入

渡部 卓雄[1,a)]

概要：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 (FRP) 言語 Emfrp のための文脈指向プログラミング機構を提案する．提案方式では時変値の値にもとづく層の活性化機構を導入し，実行時文脈の変化に伴う振る舞いの動的な変更を FRP の枠組み内で宣言的に記述できるようにしている．これにより，小規模組込みシステムにおける適応動作のモジュール化が可能になる．

キーワード：関数リアクティブプログラミング, 文脈指向プログラミング, 小規模組込みシステム

Introduction of a Context-Oriented Programming Mechanism to an FRP Language for Small-Scale Embedded Systems

Takuo Watanabe[1,a)]

Abstract: This paper briefly describes the design of a simple context-oriented programming extension to Emfrp, a purely functional reactive programming language for resource-constrained embedded systems. The proposed extension provides a layer mechanism along with a feature to describe events along with the layer activation. An example shows that the extension can eliminate typical cross-cutting code that often appear in plain Emfrp programs.

Keywords: Functional Reactive Programming, Context-Oriented Programming, Small-Scale Embedded Systems

1. はじめに

本研究は，小規模組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への文脈指向プログラミング機構の導入が，(a) 適応動作記述におけるモジュール性の改善，および (b) 適応動作の安全性・信頼性向上に寄与することを明らかにすることを目的とする．本稿では上記目的 (a) に向けて，筆者らが開発している言語 Emfrp への文脈指向プログラミング機構の導入について概要を述べ，例題を通して同機構の有用性について議論する．

一般に組込みシステムは，外界から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**リアクティブシステム**である．リアクティブシステムに対する入力は，離散的なイベントや連続的な値の変化として与えられるが，その順序やタイミングはあらかじめ予測できない．そのため，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，ポーリングなどのパターンやイディオムが多用される．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や再利用性の低下につながる．

リアクティブプログラミングは，**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によっ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 [1]．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，マイクロコントローラに代表される小規模組込みシステムを対象とした純粋関数型言語 Emfrp [4] を設計・実装し，いくつかの例題を通してその有用性を明らかにしてきた．Emfrp はマイクロコントローラ等，資源の限られた環境で実行することを前提として設計されている．

[1] 東京工業大学 情報理工学院 情報工学系
Department of Computer Science, School of Computing, Tokyo Institute of Technology
[a)] takuo@acm.org

具体的には静的型付けに加え，時変値を一級データではなく名前で参照することや再帰の禁止といった言語設計上の制約を設けている．これにより，プログラムが利用する記憶領域の大きさは静的に決定できる．また実行系としてはpush 型と呼ばれる方式を採用しており，生成されるコードサイズを抑えている．

以上のような設計方針にもとづき，Emfrp は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記述に適した言語機構を提供している．しかし上で述べた言語設計上の制約により，実行時情報にもとづく柔軟な振る舞いの表現に制約が生じる．例えば組込みシステムの実行において重要な，変化する環境への適応動作を適切にモジュール化して記述することが難しい．

このような問題に対処することを目的として，筆者らは Emfrp における自己反映計算 (reflection) の機構を提案し，ロボットの適応動作記述への応用等を通してその有効性を示した [7]．この自己反映計算機構では，Emfrp の実行機構を Emfrp 自身で記述したメタレベルモジュールを導入し，かつオブジェクトレベルとメタレベルの相互作用を時変値を介して行うようにしている．これにより，新たな言語機構を導入することなく純粋な FRP 言語の枠組み内で自己反映動作の記述が可能となる．

これによって適応動作の宣言的な記述が可能になったが，その記述はメタレベルモジュール内で行うため，オブジェクトレベルで記述されるアプリケーション動作とは分離される．しかし必ずしも汎用的ではない，アプリケーションに特化した適応動作を記述したい場合，アプリケーションと完全に分離した記述が必要になることで却って記述性や可読性の低下につながることがある．

適応的な動作のモジュール化を促進するプログラミングパラダイムとして**文脈指向プログラミング** (Context-Oriented Programming, COP) [2] がある．一般的な自己反映計算機構とは異なり，文脈（実行時情報）に依存した処理のモジュール化に特化した言語機構を導入することで，適応動作を見通しよく記述することを可能にする．多くの COP 言語は，Java などの既存の言語に文脈に依存する処理を分離して記述するための**層** (layer) と呼ばれる言語機構，および実行時情報に依って適切な層を活性化する実行時機構によって実現されている．現在までに，組込みシステム，特にロボットのように移動することで周囲の環境が変化するようなシステムにおける COP の有用性が示されている [3,5]．

本稿では，組込みシステムに特化した言語である Emfrp への COP 機構（層と層の活性化機構）の導入が，実行時文脈に依存した処理の宣言的な記述および適切なモジュール化を可能にすることを例題を用いて概説する．提案機構の詳細は [6] を参照されたい．

```
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 fan = di >= th
13
14 # threshold
15 node th = 75.0
```

図 1 温度・湿度センサによるファンの制御プログラム

2. Emfrp

Emfrp [4] は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 Emfrp の設計と実装の概略について例を通して説明する．

Emfrp のプログラム例として，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[*1]を計算し，その値が規定値以上になっている間ファンのスイッチを ON にする．

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（1–5 行目）ではモジュール名や入出力ノードおよび使用するライブラリが宣言され，本体（6 行目以降）では型，定数，関数およびノード（後述）が定義される．

ノードは Emfrp における時変値を表す言語機構であり，その値は時刻と共に変化し得る．この例では `tmp` および `hmd` が温度・湿度センサの現在の測定値に相当する．ノード `di` は不快指数を表しており，この値も `tmp` および `hmd` の値の変化に応じて自動的に再計算され変化する．さらにノード `fan` は `di` が閾値 `th` 以上であるか否かを真偽値として表しており，この値も `di` の値に応じて変化する．

Emfrp のノードは入力ノード，出力ノード，内部ノードの 3 種類に分類される．入力・出力ノードは外部機器に接続されている．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・湿度センサおよびファンのスイッチ（GPIO ポート）に接続される．内部ノードはそのような外部機器との接続を持たないノードで，この例では `di` と `th` が内部ノードであり，それぞれ不快指数とファン制御のための閾値を表している．

入力ノードの値は接続されている外部機器によって決まるが，それ以外のノードはモジュールの本体において予約

[*1] 気温を t (°C)，湿度を h (%) としたときに $0.81t+0.01h(0.99t-14.3)+46.3$ で与えられる値で，75 以上になると多くの人が蒸し暑さで不快感を感じるとされている．

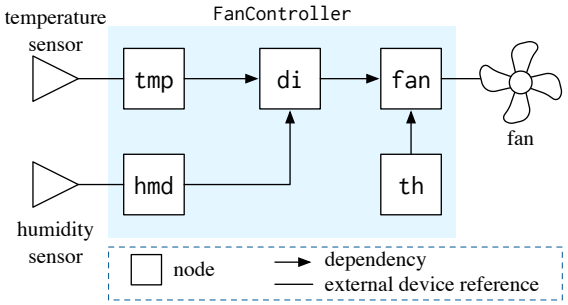

図 2 図 1 のグラフ表現

語 node を用いて定義される．図 1 では 8-9 行目，12 行目，15 行目がそれぞれ di, fan, th の定義である．各ノードの値は=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．図 2 は図 1 におけるノード間の依存関係をグラフで表現したものである．Emfrp におけるプログラムの実行は，このグラフに沿って入力ノードから出力ノードに向かって値を更新しつづけることとして実現される．これにより，入力ノードに接続された外部機器（この例では 2 つのセンサ）の値の変化が出力ノードに伝搬する．そのため，ノードの依存関係はサイクルを含まない有向グラフ (DAG) となる必要がある．

Emfrp を含む FRP 言語では，上記のように時変値間の関係を式として記述することでリアクティブな動作（イベント駆動的な動作）を宣言的に記述する．特に組込みシステムにおいては，ポーリングループや割り込みハンドラ等，プログラムの見通しを低下させる要因を排除できる．

3. ファン制御プログラムへの機能追加

図 3 は図 1 にいくつかの機能を追加したものである．具体的には，(a) 不快指数 di の値が閾値 th 近辺で変化した際にファンのスイッチが頻繁に ON/OFF を繰り返さないようにするヒステリシス制御，(b) ファンが現在までに ON になった合計時間を LCD に表示する機能，および (c) ファンのモーターを保護するため連続して 2 時間以上 ON になった際に強制的に 30 分間 OFF にする機能である．紙面の都合で詳細は省くが，(a)〜(c) の各機能はそれぞれ 19 行目，22–29 行目，32–47 行目で実装されている．また (c) については fan の定義の変更（14–16 行目）も必要となる．

ここで 19 行目における fan@last などのように，ノード名の後に@last を伴う式は**直前の時点**におけるノードの値（直前値）を表している．直前値の参照を伴うノードについては，定義する際に 14 行目などのように init を用いてプログラムの実行開始時点における直前値を指定する必要がある．直前値を用いることで状態に依存する動作を記述できるようになる．例えば 19 行目ではヒステリシス制御のための閾値に対するオフセットの値をファンの状態に応じて変更している．

```
1  module FanController2  # module name
2  in  tmp   : Float,     # temperature sensor
3      hmd   : Float,     # humidity sensor
4      clock : Int        # POSIX time system clock
5  out fan   : Bool,      # fan switch
6      ctime : Int        # LCD for time
7  use Std                # standard library
8
9  # discomfort (temperature-humidity) index
10 node di =
11     0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
12
13 # fan switch
14 node init[False] fan =
15     if timer@last > 0
16     then False else di >= th
17
18 # threshold with hysteresis offset
19 node th = 75.0 + if fan@last then -0.5 else 0.5
20
21 # cumulative operating time of fan
22 node init[0] ctime =
23     if fan@last
24     then clock - ctime0 else ctime@last
25
26 # base time for ctime (See Figure 3)
27 node init[0] ctime0 =
28     if !fan@last && fan
29     then clock - ctime@last else ctime0@last
30
31 # continuous operating time of fan
32 node init[0] otime =
33     if fan@last
34     then clock - otime0 else otime@last
35
36 node init[0] otime0 =
37     if !fan@last && fan
38     then clock else otime0@last
39
40 # timer for shutdown time
41 node init[0] timer =
42     if (otime@last > 7200) || (timer@last > 0)
43     then timer0 - clock else timer@last
44
45 node init[0] timer0 =
46     if !(otime@last > 7200) && (otime > 7200)
47     then clock + 1800 else timer0@last
```

図 3 機能を追加したファン制御プログラム

図 3 では，至る所に条件式 (if式) が存在し，その条件として同じ式が多く出現している．例えば fan@lastという条件式は，19, 23, 33 行目に出現している．実行時文脈（例えばファンが ON になっている文脈やモーター保護タイマーが有効である文脈等）に依存する動作は，このように条件式を伴って記述されることになる．しかし文脈に依存する範囲が広くなることで，同様の条件を表す式がプログラム中に散在することになる．このような状況は横断的関心事とみなすことができ，プログラムのモジュール性・可読性を低下させる．

```
1  module FanController2  # module name
2  in  tmp   : Float,     # temperature sensor
3      hmd   : Float,     # humidity sensor
4      clock : Int        # POSIX time system clock
5  out fan   : Bool,      # fan switch
6      ctime : Int        # LCD for time
7  use Std                # standard library
8
9  # discomfort (temperature-humidity) index
10 node di =
11     0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
12
13 # fan switch
14 node init[False] fan = di >= th
15
16 # threshold
17 node th = 75.0
18
19 # hysteresis behavior
20 layer HYSTERESIS where fan
21   enter node th = @proceed - 0.5
22   exit node th = @proceed + 0.5
23
24 # cumulative operating time
25 layer CTIME where fan
26   enter node init[0] ctime0 = clock - ctime@last
27   retain node init[0] ctime = clock - ctime0
28
29 # continuous operating time
30 layer OTIME where fan
31   enter node init[0] otime0 = clock
32   retain node init[0] otime = clock - otime0
33
34 # stop the fan for 30 minutes
35 # after 2 hours continuous operation
36 layer SHUTDOWN where (otime > 7200 || timer > 0)
37   enter node init[0] timer0 = clock + 1800
38   retain node init[0] timer = timer0@last - clock
39   node fan = False
```

図 4 COP 拡張を用いたファン制御プログラム

4. COP 拡張

前節で述べた問題点を解決するために，文脈指向プログラミング (COP) 機構を Emfrp に導入する．具体的には以下のような層 (layer) を記述する構文を追加する．

layer L where e

ここで L は層の名前，e は条件式である．この行に続いてノードの定義が続く．条件 e が成立したとき，言い換えると e で規定される文脈に入ったときに L は活性化され，続くノードの定義がオリジナルの定義を上書きする．このようにして文脈に依存する動作を層として記述できる．

図 4 は，図 3 を COP 拡張を用いて書き直したものである．前節で導入した拡張 (a) は層 HYSTERESIS で，(b) は層 CTIME で，(c) は層 OTIME と SHUTDOWN でそれぞれ実現されている．それぞれの層は個別に導入や削除が可能であり，層の独立性は高い．また図 3 とは異なり，ノード fan の定義への変更は不要である．つまり拡張部分（層の定義）以外におけるノードの定義（10–17 行目）は拡張前の図 1 とほぼ同等である．

この例における層に付随するノード定義には **enter**, **exit**, **retain** というキーワードが付いている．これらはそれぞれ層が活性化された時 (**enter**) と非活性化された時のみ更新 (**exit**,) されること，および層が活性化されていないときは最後の値を保持すること (**retain**) を表している．これらのような，層な活性・非活性化に連動するノードの定義の導入により，文脈に依存する様々な振る舞いを条件式等を多用せずに簡潔に記述できるようになっている．

5. まとめ

本稿では，組込みシステムに特化した関数リアクティブプログラミング (FRP) 言語である Emfrp への文脈指向プログラミング (COP) の機構の導入について概要を述べた．文脈に依存した振る舞いに相当するノードの定義を層としてモジュール化することに加え，層が活性化したときの振る舞いを制御する機構を導入することで，実行時文脈に依存した振る舞いの宣言的かつ簡潔な記述および適切なモジュール化を可能にすることを例題を用いて概説した．

提案した COP 拡張は，拡張前の Emfrp への変換として実現可能である．しかしその際に，層を変換することで誤ったノードの依存関係（サイクル）が導入されることがある．その解決は今後の課題の一つである [6].

謝辞

本研究は JSPS 科研費 18K11236 の助成を受けている．

参考文献

[1] Bainomugisha, E., Carreton, A. L., Van Cutsem, T., Mostinckx, S. and De Meuter, W.: A Survey on Reactive Programming, *ACM Computing Surveys*, 45(4), doi:10.1145/2501654.2501666 (2013).
[2] 紙名哲生: 文脈指向プログラミングの要素技術と展望, コンピュータソフトウェア, 31(1), pp. 3–13, doi:10.11309/jssst.31.1_3 (2014).
[3] 佐伯優太, 谷川郁太, 久住憲嗣, 福田晃: ContextROS: ロボットオペレーティングシステムへのコンテキスト指向プログラミングの適用, 組込みシステムシンポジウム (ESS 2017), pp. 47–53, http://id.nii.ac.jp/1001/00183029/ (2017).
[4] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, CROW 2016, ACM, pp. 36–44, doi:10.1145/2892664.2892670 (2016).
[5] Watanabe, H., Sugaya, M., Tanigawa, I., Ogura, N. and Hisazumi, K.: A Study of Context-Oriented Programming for Applying to Robot Development, COP 2015, ACM, doi:10.1145/2786545.2786551 (2015).
[6] Watanabe, T.: A Simple Context-Oriented Programming Extension to an FRP Language for Small-Scale Embedded Systems, COP 2018, ACM, doi:10.1145/3242921.3242925 (2018).
[7] Watanabe, T. and Sawada, K.: Towards Reflection in an FRP Language for Small-Scale Embedded Systems, LASSY 2017, ACM, doi:10.1145/3079368.3079387 (2017).