Vol.2022-EMB-60 No.1
2022/7/8

入出力の動的な切替機構をもつ組込みシステム向けFRP言語の検討

瀧本 哲史[1,a)]　森口 草介[1,b)]　渡部 卓雄[1,c)]

概要：関数リアクティブプログラミング (FRP) は，時間変化する値 (時変値) を組み合わせて反応的な振る舞いを記述するプログラミングパラダイムであり，組込みシステムはその応用例の一つである．組込みシステムにおいて，使用しないセンサの電源を切るなど，状況によって使用する入出力デバイスを切替えたい場合がある．しかし，従来の FRP ではそのような外部デバイスの有効無効を切替える表現が困難であった．その課題を解決するため，本研究では FRP 言語内で入出力の動的な切替えの表現を可能にする言語機構を検討する．

1. はじめに

組込みシステムの多くは外部からの入力に応じて反応的に状態や出力を変化させる．そのようなシステムは特に**リアクティブシステム**と呼ばれる．**関数リアクティブプログラミング（FRP）**は，**時変値**という時間変化する値の組み合わせで宣言的にリアクティブシステムを記述するプログラミングパラダイムである．これまで Emfrp[1] や Hailstorm[2] といった組込みシステム向け FRP 言語が開発され有用性が示されてきた．

本研究の議論は Emfrp の switch 拡張 [3] の後続である**XStorm**[4] を議論の対象とする．Emfrp の switch 拡張及び XStorm は，Emfrp の持つメモリ使用量を静的に決定できる性質を維持しつつ，状態に応じて時変値間の関係式を切り替えることができる特徴を持つ．これにより，状態に依存して動作を変える組込みシステムの記述を見通しよく行うことができる．

組込みシステムでは状況に応じて使用する入出力デバイスを切り替えたい場合がある．具体的には，省電力化のため使用しないデバイスの電源を一時的に切ることなどがそれにあたる．しかし，入出力のインターフェースが固定的である XStorm では，そのような入出力デバイス自体の有効無効を切り替える表現は困難である．

[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, Meguro, Tokyo 152-8552, Japan
[a)] takimoto@psg.c.titech.ac.jp
[b)] chiguri@acm.org
[c)] takuo@acm.org

そこで，本研究では XStorm をもとにした FRP 言語内で入出力の動的な切替の表現を可能にする言語機構について提案する．そしてその言語機構の実装方法についても検討する．

本論文では，まず第 2 節で XStorm の概要を述べ，第 3 節では XStorm で動的な入出力デバイスの切替を実装する際に生じる問題点を挙げる．第 4 節で本研究で提案する言語機構の説明及び前述の問題点への有効性を示したのち，第 5 節でその言語機構の実行モデルも示す．第 6 節では本研究と関連研究の比較を行い，最後に第 7 節で結論と今後の課題について述べる．

2. XStorm

XStorm は，小規模組込みシステム向け FRP 言語 Emfrp をベースとした FRP 言語である．Emfrp には状態依存動作の簡潔な記述を可能にした switch 拡張が提案されており，その後続として同じ著者が XStorm を提案した．両者の言語機構はほぼ同じだが，XStorm はより効率の良い実行モデルを持っている．

XStorm は状態に応じて動的に時変値間の関係式を切り替えることができる特徴をもつ．例として，XStorm によるストップウォッチプログラムを**図 1** に示す．`StopWatch`モジュールは，1 秒毎のパルス `pulse1s`及びスタートストップボタンとリセットボタンが押された時に立つフラグ `startStop,reset`を入力とし，現在の計測時間 `time`を出力する（2,3 行目）．XStorm では時変値をノードと呼び，例えば `node time = if reset then 0 else time@last`のように定義される（7 行目）．この部分では，ノード `time`の値が `if reset then 0 else time@last`という関係式で計算されるこ

```
1  switchmodule StopWatch {
2    in  pulse1s: Bool, startStop(False): Bool, reset:
       Bool
3    out time(0): Int
4    init Stop
5
6    state Stop {
7      out node time = if reset then 0 else time@last
8      switch:
9        if (startStop && !startStop@last)
10         then Run else Retain
11   }
12   state Run {
13     out node time =
14       time@last + if pulse1s then 1 else 0
15     switch:
16       if (startStop && !startStop@last)
17         then Stop else Retain
18   }
19 }
```

図 1 XStorm で記述されたストップウォッチプログラム
Fig. 1 A stopwatch program in XStorm

とを表す．式中の time@lastの部分はノード timeの更新前の値（**直前値**）を表す．XStorm は関係式に従ってノードの値を更新することを繰り返す実行モデルを持っている．ノードの現在の値（**現在値**）の他に直前値を参照できるようにすることで，ノードの状態に依存した動作を記述できるようになっている．なお，その繰り返しの単位を XStorm では**イテレーション**と呼ぶ．XStorm では，switch モジュールを用いて状態に応じて動作を変える挙動を表現できる．switch モジュールの stateブロック内では各状態におけるノードの関係式に加え，switch節で次の状態を指定する（6-11,12-18 行目）．

XStorm プログラムでは再帰関数や再帰データ型の禁止，ノードの過去の値の参照を直前値だけに限るといった，メモリ量を抑えるような制約がある．この制約によって，組込みシステムのような限られたメモリ環境下でも反応性の高い動作を実現している．

XStorm プログラムは C++プログラムにコンパイルされる．生成されたコードの主となる関数は activateであり，これを呼び出すことで XStorm プログラムが実行される．生成されたプログラムには外部関数 input,outputも含まれ，ユーザーがそれらを実装する必要がある．inputでは XStorm プログラムのメインとなるのモジュールへの入力値を入力デバイスから読み取って設定することが想定されている．一方 outputではメインのモジュールの出力ノードの値を出力デバイスに設定することが想定されている．**図 2** に図 1 の XStorm プログラムの利用例を示す．inputや output関数で各入出力デバイスとの入出力処理を行い，メイン関数で入出力デバイスの初期化後 activate()を呼び出している．

```
1  void input(int* pulse1s, int* startStop, int* reset) {
2    *pulse1s = readOscillator();
3    *startStop = readButton(button1);
4    *reset = readButton(button2);
5  }
6  void output(int* time) {
7    displayTime(*time);
8  }
9  int main(void) {
10   setUpOscillator(); setUpButton(button1);
11   setUpButton(button2); setUpDisplay();
12   activate();
13 }
```

図 2 XStorm プログラムから生成される C++プログラムの利用例
Fig. 2 A client C++ program of the stopwatch program

3. FRP における入出力デバイス切替の問題点

本節では本研究の動機となる FRP における入出力デバイスの切替に関する問題点を例題の記述を用いて説明する．

例題として扇風機制御システムの実装を考える．このシステムは現在の気温が 28 度以上の時に扇風機を稼働させる．また人感センサを用いたスリープ機能を搭載しており，人がいないときは省電力化のため扇風機は稼働させず温度センサの電源も切ることにする．

このシステムを XStorm を用いて記述することを考える．XStorm ではノードの関係式を動的に切替えることはできるが，入出力インターフェースは固定的で動的に入出力ノード自体の追加・削除はできない．そのため，XStorm プログラムでは温度センサに対応する入力ノードがなくなることを直接的に表現できない．そこで XStorm プログラムから生成される C++コードの関数 inputで温度センサの管理を行うようにする．

図 3，**図 4** に前述の方針で扇風機制御システムを実装した XStorm プログラム及びその利用コードを示す．FanControllerモジュールは入力として人がいるかを示す humanと現在の気温 tmpをとり，扇風機の稼働させるか否かを表す fanを出力する．状態は Runと Sleepの 2 つあり，人感センサの値によってお互いに状態遷移する．Runでは現在の気温が 28 度以上で扇風機が稼働させる一方，Sleepでは常に扇風機を稼働させないよう定義されている．input関数では FanControllerモジュールへの入力に人感センサの値と温度センサの値が設定されている．さらに，現在及び直前の人感センサの値を用いて温度センサの電源管理を同時に行っている．温度センサが稼働していない場合には現在の気温 tmpの値を設定していない．

この実装には問題点が複数ある．一つ目は，入力デバイスの切替を含めたシステムの仕様が XStorm と C++の二つのプログラムに分散してしまった点である．システムの仕様の修正等の際にそれら二つのプログラムを扱う必要があり，メンテナンス性を下げている．例えば，システムの

```
1  switchmodule FanController {
2    in  human: Bool, tmp: Float
3    out fan: Bool
4    init Run
5
6    state Run {
7      out node fan = tmp >=. 28.0
8      switch: if !human then Off else Retain
9    }
10   state Sleep {
11     out node fan = False
12     switch: if human then On else Retain
13   }
14 }
```

図 3 XStorm による扇風機制御システムの実装
Fig. 3 The fan controller system implementation in XStorm

```
1  switchmodule FanController with
2    in  human: Bool, tmp: Float
3    out fan: Bool
4  {
5    init Run
6    state Run with human, tmp, fan {
7      out node fan = tmp >=. 28.0
8      switch: if !human then Off else Retain
9    }
10   state Sleep with human, fan {
11     out node fan = False
12     switch: if human then On else Retain
13   }
14 }
```

図 5 Cios による扇風機制御システムの実装
Fig. 5 The fan controller system implementation in Cios

```
1  int currHuman = 0, prevHuman = 0;
2
3  void input(int* human, double* tmp) {
4    prevHuman = currHuman;
5    *human = currHuman = readHumanDetector();
6    if (currHuman) {
7      if (!prevHuman) turnOnThermometer();
8      *tmp = readThermometer();
9    } else {
10     if (prevHuman) turnOffThermometer();
11   }
12 }
```

図 4 XStorm による扇風機制御システムプログラムの利用例
Fig. 4 A client program of the fan controller program in XStorm

状態遷移の条件を変更することを考える．すると，XStorm プログラムだけでなく C++プログラム中の入出力デバイスを切り替えるロジックまで修正する必要がある．また，C++プログラム中の入出力管理ロジックを誤って XStorm プログラムの状態遷移と整合性のないものにしてしまう可能性など，バグを埋め込む余地ができてしまっている．二つ目は，XStorm プログラムで誤って無効な値を参照してしまう可能性がある点である．XStorm プログラムには入出力ノードの値の有効無効が現れない．そのため，この例題における電源が切れた温度センサの値といったような無効な値の参照を防ぐことができない．この問題はシステムにバグや脆弱性を引き起こす可能性がある．

4. 提案手法とその有効性

第 3 節で述べた問題点を解決するため，使用の有無を切り替えることのできる時変値など新たな言語機構を XStorm に導入した言語 Cios を提案する．Cios は各モジュール・状態が使用する外部入出力を明示させるような記法になっており，状態ごとに異なる外部入出力の使用を指定することで，動的な入出力の切替を FRP 言語で表現できるようにした．

外部入出力ノードは入出力デバイスと対応するノードであり，XStorm におけるメインのモジュールの入出力ノードにあたる．重要な点として，外部入出力ノードは各イテレーションごとに有効無効を切り替えることができる．これは，外部入出力を使用するか否かを変化させるという概念を時変値に加えたオブジェクトとして考えることができる．外部入出力ノードが無効なイテレーションでは対応する入出力デバイスの入出力処理は行わない．そのため外部入出力ノードは直前値を持たない場合があるので，現在値のみの参照に制限している．外部入出力ノードの有効無効は関係式での束縛や参照，そして**使用記述**によって定まる．使用記述は，switch モジュールの各状態において有効にしたい外部入出力ノードを明示的に記述したものである．各状態に異なる使用記述を与えることで動的な入出力切替を表現することができるようになった．さらに，使用記述で指定されなかった外部入出力ノードはスコープから外すようにし，無効な入出力への参照や束縛の防止も実現した．**図 5** に第 3 節で挙げた扇風機制御システムの Cios での実装を示す．外部入出力ノードは入出力ノードと区別して switch モジュールの宣言の冒頭の `with`キーワードにつづけて宣言する（2,3 行目）．使用記述は `state`ブロック冒頭に記述する．図 5 では，例えば状態 `Sleep`の使用記述では外部入出力ノードは `with human, fan`となっており，`tmp`が指定されていない（10 行目）．これによって状態 `Sleep`では温度センサを使用しないことを示している．使用記述には**表 1** に示すように 4 つの形式があり，それぞれ意味が異なる．例えば状態 `Sleep`の使用記述は代わりに `without tmp`と書くこともできる．使用記述が `with _, without _`の場合は，その状態で参照・束縛される外部入出力ノードを有効にすると推論される．

Cios プログラムは XStorm プログラムと同様 C++プログラムにコンパイルされる．生成コードの変更点として，`input`関数や `output`関数を外部入出力ノードごとに分けたほか，新たに**フック**を含めるようにした．フックは各外部

表 1　使用記述の形式
Table 1　Available usage annotations

使用記述	有効な外部入出力ノードのリスト
なし	$\mathcal{E}$
with ε	ε
without ε	$\mathcal{E} \setminus \varepsilon$
with _, without _	推論される

* $\mathcal{E}$ は状態が属する switch モジュールがとる外部入出力ノードのリスト

```
1 double input_tmp(void) { return readThermometer(); }
2 void initialize_tmp(void) { turnOnThermometer(); }
3 void terminate_tmp(void) { turnOffThermometer(); }
4
5 int input_human(void) { return readHumanDetector(); }
6 ...
```

図 6　Cios による扇風機制御システムプログラムの利用例
Fig. 6　A client program of the fan controller program in Cios

入出力ノードの有効無効が切り替わった時に Cios ランタイムに自動的に呼び出される関数である．**図 6** に Cios による扇風機制御システムのプログラムの利用コードを示す．例えば関数 `initialize_tmp`は外部入出力ノード `tmp`の状態が無効から有効に切り替わった時に呼び出されるフックである．逆に関数 `terminate_tmp`は外部入出力ノード `tmp`の状態が有効から無効に切り替わったときに呼び出されるフックである．フックの用途は入出力デバイスの初期化処理と終了処理を想定している．Cios ランタイムが適切なタイミングで自動的にフックを呼ぶため，ユーザーは入出力デバイスの切替ロジックを記述する必要はない．単にデバイス単体の初期化処理・終了処理をそれぞれ記述するだけで良くなり，第 3 節の実装と比較してメンテナンス性が上がっていることがわかる．

Cios で導入した言語機構はいわゆるウォーミングアップの表現も可能にしている．デバイスの電源を入れるなどの初期化処理・終了処理は時間がかかることがあり，システムの応答性向上のため意図的にデバイスの電源をつけたままにしたい場合がある．Cios では，関係式等から参照されない外部入力ノードを意図的に使用記述で指定することで，それに対応するデバイスのウォーミングアップを表現することができる．ただし，束縛されていない外部出力ノードを意図的に使用記述に含めることはできないようにしている．これは外部出力ノードの値つまり出力デバイスに設定するべき値が未定義になってしまうのを防ぐためである．

5. 実行モデル

Cios の実行モデルについて，まずベースとなる XStorm の実行モデルについて説明し，その後外部入出力ノードなどの導入に伴う実行モデルの変更点について述べる．

第 2 節でも述べたように，XStorm は各ノードの現在値の更新計算を行うイテレーションを反復的に行うことでリアクティブな動作を実現している．コンパイル時にノード間の依存関係から更新順序が適切に決定され，その順序に従って実行時にノードの現在値の更新計算が行われていく．なお入力ノードの現在値には入力デバイスからサンプリングされた値が設定される．switch モジュールでは，モジュール内全てのノードの更新が終わった後にさらに状態の更新計算が行われる．

Cios の実行モデルは XStorm のものとほとんど同じであるが，各イテレーションの冒頭で外部入出力ノードの使用の有無の判定とフックの呼び出し処理を追加で行うようにした．現在の状態における各外部入出力ノードの使用の有無の判定方法はコンパイル時に次のように決定される．

- モジュールや現在の状態においてその外部入出力ノードが参照または束縛される場合，使用有りと判定
- 現在の状態の使用記述でその外部入力ノードが指定されている場合，使用有りと判定
- それ以外は使用無しと判定

なお，使用の有無が常に一定であるとコンパイル時に決定できる外部入出力ノードについては，各イテレーションでの使用の有無の判定を行わない最適化を行うことができる．フックは現在のイテレーションと直前のイテレーションにおける使用の有無に応じて適宜呼び出される．`initialize_`から始まるフックについては，対応する外部入出力ノードが現在のイテレーションで使用有りで直前では使用無しとなっていた場合に呼び出す．逆に `terminate_`から始まるフックについては，対応する外部入出力ノードが現在のイテレーションで使用無しで直前では使用有りとなっていた場合に呼び出す．

Cios と XStorm で同じシステムを実装した際に，実行モデルの違いによる性能差はほぼないと考えられる．なぜなら，今まで XStorm の `input`関数内で行っていた入出力デバイス管理処理を Cios のランタイムが単に代わりに行うようになっただけだからである．実際に例題の扇風機制御システムの XStorm 実装と Cios 実装の性能を比較したところ，実行時間は同程度だった（**表 2**）．ただし実行時間の測定は以下のように行った．

- 測定環境: MacBook Air (M1, 2020), macOS Monterey 12.4
- C コンパイラ: Apple clang 13.1.6 (arm64-apple-darwin21.5.0)
- 同じダミーの入力を与えて 10^8 回イテレーションを行うのにかかる時間をそれぞれ計測

6. 関連研究

Hailstorm はリソースが限られた IoT デバイスを対象とする FRP 言語である．Hailstorm は FRP の中でも Arrowized FRP（AFRP）[5] というパラダイムを持ってい

表 2　扇風機制御プログラムの性能比較
Table 2　Performance comparison of the fan controller system

	実行時間（秒）
XStorm	2.05 ± 0.01
Cios	2.14 ± 0.01

る．AFRP では，時変値間の関係式は記述するのではなく時変値上の関数（シグナル関数）を組み合わせていく．Hailstorm では `switch#`という動的な振る舞いの切替を可能にする組込みの関数が提供されている．`switch#`の第一引数は，振る舞いの切替条件となる値を出力するシグナル関数である．`switch#`の第二引数は第一引数の出力を受けてシグナル関数を返す関数であり，ここで条件に応じて違うシグナル関数を返すことで動的に動作を切替えることができる．しかし，Hailstorm では条件に応じて違う入出力を使用するシグナル関数を返せない問題がある．Hailstorm では，外部入出力を AFRP で安全に扱うため，入出力リソースを表すシグナル関数にそれぞれ固有の型がつくようになっている．その結果，条件に応じて違う入出力を使用しようとすると型に不整合が起きエラーになってしまう．この問題により，Hailstorm では動的な入出力デバイスの切替を表現できない．

Emfrp には文脈指向プログラミング (Context-Oriented Programing) を導入する拡張が提案されている [6]．ある条件が満たされた時（文脈）に起こしたい振る舞いを層というオブジェクトで定義し，実際に条件が満たされた場合にその層を有効にするという手法で動的な振る舞いの変化を記述する．しかし，その拡張では層にローカルな入出力ノードのようなものは提供されていないため，動的な入出力デバイスの切替を表現できない．また，各層は互いに独立であり層の中の定義は他の層からは見えないようになっている．これにより，いくつかの異なる層で共通の入出力デバイスを用いるといったことができない．

Statecharts[7] や MATLAB/Simulink の Stateflow[8] はシステムの状態遷移をグラフィカルに表現するツールである．特に Stateflow は組込みシステム開発でも行われている Model-Based Development（MBD）の文脈でよく用いられるものである [9]．XStorm における状態の表現は Statecharts を参考にしており，結果として Cios も Statecharts と類似している部分がある．Stateflow では，シンボルの有効無効を状態に応じて切り替えるような機能は提供されていない．代わりに，Statecharts や Stateflow ではアクションという状態への出入りや状態遷移の際に実行するコードを記述する方法が用意されている．Cios のフックは状態遷移時のアクションに対応していると見ることができる．

7. 結論

本研究では，FRP 言語において動的な入出力デバイスの切替を可能にする言語機構を提案し，その有効性を示した．提案した言語機構は各状態において使用する外部入出力を明示させる記法になっており，これにより宣言的に入出力デバイスを切り替えるロジックを表現できる．

Cios の構文は switch モジュールにおける各状態やそこで有効な外部入出力ノードを抽出しやすいものとなっている．それを用いて，例えば Cios プログラムからクリプキ構造のようなものを生成し入出力デバイスについてのモデル検査を行うことを考えられる．

Cios にはいくつか問題点がある．まず，Cios では外部入出力ノードは有効無効の 2 状態のみしか表現できない．従って，オン・オフ・スリープのような 3 つ以上の状態を Cios で表現することができない．この問題については，外部デバイスが取りうる状態をユーザーが定義できるような言語機構の追加を考えている．また，Cios では外部入出力ノードを組み合わせたり値を加工したりして新たな外部入出力ノードを作ることができない．この機能は，例えば既存の Cios モジュールをラップして複合センサに対応させたい場合に必要である．また，入出力値を安定させるために直近のいくつかの値の平均値をとるといったユースケースもある．この問題についても，既存の外部入出力ノードを元に仮想的な外部入出力ノードをユーザーが定義できる言語機構を追加することで解決することを考えている．

謝辞　本研究の一部は JSPS 科研費 21K11822 および 19K20245 の助成を受けている．

参考文献

[1] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *Companion Proceedings of the 15th International Conference on Modularity*, MODULARITY Companion 2016, Association for Computing Machinery, pp. 36–44 (online), DOI: 10.1145/2892664.2892670 (2016).
[2] Sarkar, A. and Sheeran, M.: Hailstorm: A Statically-Typed, Purely Functional Language for IoT Applications, *Proceedings of the 22nd International Symposium on Principles and Practice of Declarative Programming*, PPDP '20, Association for Computing Machinery, (online), DOI: 10.1145/3414080.3414092 (2020).
[3] 松村有倫, 渡部卓雄：組込みシステム向け FRP 言語における状態依存動作のための抽象化機構，情報処理学会論文誌プログラミング（PRO），Vol. 13, No. 2, pp. 1–13 (2020).
[4] 松村有倫：組込みシステム向け FRP 言語における状態依存動作のための抽象化機構，修士論文，東京工業大学 情報理工学院 (2020).
[5] Courtney, A. and Elliott, C.: Genuinely Functional User Interfaces (2001).
[6] Watanabe, T.: A Simple Context-Oriented Programming Extension to an FRP Language for Small-Scale Embedded Systems, *Proceedings of the 10th International*

Workshop on Context-Oriented Programming: Advanced Modularity for Run-Time Composition, COP '18, Association for Computing Machinery, pp. 23–30 (online), DOI: 10.1145/3242921.3242925 (2018).

[7] Harel, D.: Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Program.*, Vol. 8, No. 3, pp. 231–274 (online), DOI: 10.1016/0167-6423(87)90035-9 (1987).

[8] MathWorks: Stateflow, MathWorks (online), available from ⟨https://www.mathworks.com/products/stateflow.html⟩ (accessed 2022-06-07).

[9] 神山達哉, 添田隆弘, 兪　明連, 横山孝典：組み込み制御ソフトウェア開発のための Simulink・UML モデル変換ツール，技術報告 7，東京都市大学, 東京都市大学, 東京都市大学, 東京都市大学 (2010).