組込みシステム向けFRP言語における状態遷移モデルに基づいた周辺装置の状態制御

瀧本 哲史　森口 草介　渡部 卓雄

小規模組込みシステム向け FRP 言語 XStorm では，状態遷移モデルに基づいて時変値間の関係を動的に変更でき，状態依存動作を簡潔に記述できる．しかし，例えば消費電力削減のため，状態に応じて使用する周辺装置を切替えるといった際に，C 言語で書かれた周辺装置のドライバーコード内で切替え処理を書く必要があった．その結果，XStorm プログラムとドライバーコードの間のロジックの不整合が検出できない．本研究では，XStorm の状態遷移モデルを発展させ，状態ごとに周辺装置の電源等の状態 (モード) を宣言できるようにすることで，FRP 言語内での周辺装置のモードの切替えも含めたロジックの記述を可能にした．これにより前述の問題を解決したほか，ランタイムによるモードの自動管理を行えるようになった．

XStorm, an FRP language for small-scale embedded systems, allows us to concisely describe state-dependent behaviors based on the state hook model. However, when we use different sets of peripheral devices depending on states, device management, such as switching power modes, should be implemented in a driver code in C. This would result in bugs as inconsistency between the state in the XStorm program and that in the driver code cannot be detected. In this research, we extend XStorm's state hook model to express modes of peripherals that depend on states. By the extension, the language manages modes of peripherals, and thus the inconsistency is statically avoided.

1　はじめに

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**) の間の関係式によって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．リアクティブシステムは外部からの入力に応じて反応的に状態遷移や出力を行うシステムであり，組込みシステムはその典型的な例である．これまでに Emfrp [11] や Hailstorm [10] といった組込みシステム向けの FRP 言語が開発され，その有用性が示されてきた．

小規模組込みシステム向け FRP 言語 XStorm [8] は，Emfrp をベースとし，状態遷移モデルに基づいて時変値間の関係式を動的に変更できる言語機構を導入した言語である．その言語機構により，XStorm では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良く行うことができる．

組込みシステムでは状況に応じて周辺機器の状態 (以後モードと呼ぶ) を切り替えたい場合がある．例として，省電力化のために使用しない周辺機器の電源モードを一時的にオフやスリープに切り替えることが挙げられる．このような，値だけでなく入出力自体の有効・無効といった状態が時間変化するような挙動は，従来の時変値の関係式のみを記述する FRP では表現が困難である．その結果，周辺機器のモード切替のロジックは FRP プログラムを駆動するプログラム (ドライバーコード) に記述する必要が出てくる．この FRP プログラムとドライバーコードへのロジックの分散は，ロジックの不整合などバグを埋め込む余地を作る．

そこで，XStorm の状態遷移モデルを発展させ，状態ごとに周辺装置のモードを明示的に指定できるような XStorm の拡張を提案する．この拡張により，FRP

Mode Management of Peripherals Based on State hook Model in FRP Language for Embedded Systems

Satoshi Takimoto, Sosuke Moriguchi, Takuo Watanabe, 東京工業大学, Department of Computer Science, Tokyo Institute of Technology.

言語内で周辺装置のモード切替も含めたロジックの表現が可能になった．さらに，モードの指定に基づいてランタイムが適切なタイミングで自動的にモード切替処理を呼び出すことができるようになった．これにより，ドライバーコード側にいつモードを切り替えるか判断するロジックを書く必要がなくなる．

我々はこれまでに，オン・オフのみのような 2 つのモードしかない場合だけを扱える言語機構をもつ FRP 言語 Cios [12] を開発した．しかし，大抵の周辺機器はそのほかにスリープモードや低電力モードを持っており，Cios でそれを扱うにはやはりドライバーコードにロジックを書くしかなかった．本論文ではモード数の制約がない言語機構を提案する．

本論文では XStorm を土台に新たな言語機構について議論するが，他の状態遷移系を陽に記述できる言語でもこの言語機構を導入できると考えている．

本論文では，まず第 2 節で XStorm の概要を述べ，続いて第 3 節では XStorm でモード切替のロジックを記述する際に生じる問題点をあげる．第 4 節では本研究で提案する言語機構の説明と前述の問題点への有効性を示す．第 5 節では関連研究との比較を行い，最後に第 6 節で結論と今後の方針について述べる．

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 [9] が提案されていた．その後，同じ著者がより効率の良い実行モデルを持つ XStorm を開発した．

図 1 は XStorm によるストップウォッチの実装である．`Stopwatch`モジュールは，1 秒ごとのパルス `pulse1s`，時間の計測の可否を決めるフラグ `stst`，計測時間をリセットするフラグ `rst`を入力とし，現在の計測時間 `time`を出力する (2-7 行目)．XStorm では時変値をノードと呼び，ノード間の関係式を記述することでロジックを構築する．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．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．関係式の中では，例えば 16 行目の `time@last`のように，`@last`アノテーションによって

```
1  switchmodule Stopwatch {
2    in  # 1秒間隔のパルス
3        pulse1s: Bool,
4        # 計測の可否を決めるフラグ
5        stst(False): Bool,
6        # 計測時間をリセットするフラグ
7        rst: Bool
8    out # 現在の計測時間
9        time(0): Int
10   # 初期状態
11   init Paused
12
13   # 時間の計測を止めている状態
14   state Paused {
15     # rstが立ったら時間をリセット
16     out node time = if rst then 0 else time@last
17     # ststがたったらMeasuringに遷移
18     switch: if stst then Measuring else Retain
19   }
20
21   # 時間を計測している状態
22   state Measuring {
23     # 1秒ごとにtimeをインクリメント
24     out node time =
25       time@last + if pulse1s then 1 else 0
26     # ststが立ったらPausedに遷移
27     switch: if stst then Paused else Retain
28   }
29 }
30
31 module Main {
32   in  pulse1s: Int, button1: Bool, button2: Bool
33   out time: Int
34
35   # 入力の立ち上がりエッジをフラグに使う
36   node stst(False) = button1 && !button1@last
37   node rst(False) = button2 && !button2@last
38   # Stopwatchモジュールをインスタンス化
39   newnode out time = Stopwatch <- pulse1s, stst,
       rst
40 }
```

図 1　XStorm で記述されたストップウォッチプログラム

```
1  void input(int* pulse1s, int* button1, int*
       button2) {
2    *pulse1s = readOscillator();
3    *button1 = readButton1();
4    *button2 = readButton2();
5  }
6  void output(int* time) {
7    displayTime(*time);
8  }
9  int main(void) {
10   setupClock(); setupDisplay();
11   setupButton1(); setupButton2();
12   activate();
13 }
```

図 2　ストップウォッチプログラムのドライバーコード

ノードの直前のイテレーションの値 (**直前値**) を参照できる．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，ステートフルなノードを宣言的に記述できる．XStorm では，switch モジュールを用いることでノード間の関係式自体を動的に変更できる．switch モジュールの `state`ブロック

内では各状態における関係式に加え，**switch**節で次の状態を指定する (17-18，26-27 行目)．なお，状態遷移が無い場合には switch モジュールの代わりに通常のモジュール (31 行目) を使える．定義したモジュールは他のモジュール内でインスタンス化できる．インスタンス化には **newnode**という構文を用いる (38-39 行目)．このストップウォッチの例では，ボタンからの入力を加工した上でインスタンス化した Stopwatchモジュールに入力している．switch モジュールを他の switch モジュール内でインスタンス化することにより階層的な状態遷移のあるシステムを簡潔に表現できる．

XStorm プログラムは C++プログラムにコンパイルされ，ユーザーは生成されたコードを駆動するドライバーコードを書く必要がある．図 2 に XStorm によるストップウォッチプログラムのドライバーコードを示す．ユーザーは，まず周辺機器とやり取りする処理をそれぞれ input関数，output関数に記述する (1-8 行目)．そして activate関数を呼び出すことで XStorm プログラムが実行される (12 行目)．

3　動機

本節では，本研究の動機となる FRP 言語における周辺機器のモード制御に関する問題点について，例題の記述を用いて説明する．

例題として，定期的に温度及び湿度を計測し Wi-Fi で送信するセンサーノードシステムを考える．このシステムはトグルスイッチによって計測をするかどうかを切り替えられる．省電力化のため，計測しない時は Wi-Fi モジュールや温度計・湿度計の電源を切る．また，トグルスイッチを入れていても，データ送信をする時以外は Wi-Fi モジュールを低電力モード，温度計・湿度計は電源オフに設定する．Wi-Fi モジュールの低電力モードでは，データ送信はできないが接続を維持するために定期的にシグナルを送る．

このシステムを XStorm で実装することを考える．図 3，4 にそれぞれ XStorm による実装とそのドライバーコードを示す．WiFiThermoHygrometerモジュールは switch モジュールを用いて定義される．状態は，温度・湿度の計測を行わない状態 Off，時間まで計測を待つ状態 Sleep，計測を行う状態 Measureの 3 つである．WiFiThermoHygrometerモジュールはトグルスイッチが入っているかを示す toggle，1 秒毎のパルス pulse1s，温度と湿度 tmp,hmdを入力にとり，送信するデータ data及び次の状態と状態遷移の組 transitionを出力する．transitionはドライバーコードにおいて周

```
1  const off = 0
2  const sleep = 1
3  const measure = 2
4  const retain = 0
5  const off2Sleep = 1
6  const sleep2Off = 2
7  const sleep2Measure = 3
8  const measure2Off = 4
9  const measure2Sleep = 5
10
11 switchmodule WiFiThermoHygrometer {
12   in  toggle: Bool, pulse1s: Bool,
13       tmp: Float, hmd: Float
14   out data: (Float, Float),
15       # 次の状態と状態遷移の組
16       # 周辺機器へのアクセスのタイミングを取ったり電源管理をしたりするため
17       transition: (Int, Int)
18   init Off
19
20   # 温度・湿度の計測を行わない状態
21   state Off {
22     out node data = (0.0, 0.0) # ダミー
23     # 温度センサがオフのためtmpの参照はバグ
24     node invalid = tmp
25     # トグルが入ったらSleepへ遷移
26     out node transition =
27       if toggle then (sleep, off2Sleep)
28       else (off, retain)
29     switch: if toggle then Sleep else Retain
30   }
31
32   # 時間まで計測を待つ状態
33   state Sleep {
34     # Sleepに遷移してきてから何秒経ったか
35     node t(0) = t@last + if pulse1s then 1 else 0
36     out node data = (0.0, 0.0) # ダミー
37     # トグルが切れたらOffへ遷移
38     # 10秒間以上
       Sleepに留まっていたらMeasureへ遷移
39     out node transition =
40       if !toggle then (off, sleep2Off)
41       else if t>=10 then (measure, sleep2Measure)
42       else (sleep, retain)
43     switch:
44       if !toggle then Off
45       else if t >= 10 then Measure
46       else Retain
47   }
48
49   # 温度・湿度を計測する状態
50   state Measure {
51     out node data = (tmp, hmd)
52     # トグルが切れたらOffへ遷移
53     # そうでなかったらSleepへ遷移
54     out node transition =
55       if !toggle then (off, measure2Off)
56       else (sleep, measure2Sleep)
57     switch: if !toggle then Off else if Sleep
58   }
59 }
```

図 3　XStorm で記述されたセンサーノードシステム

```
1  #define off 0
2  #define sleep 1
3  #define measure 2
4  #define retain 0
5  #define off2Sleep 1
6  #define sleep2Off 2
7  #define sleep2Measure 3
8  #define measure2Off 4
9  #define measure2Sleep 5
10
11 static int currState = off;
12
13 void input(
14   int* toggle,
15   int* pulse1s,
16   float* tmp,
17   float* hmd
18 ) {
19   *toggle = readToggle();
20   *pulse1s = readOscillator();
21   // measureの時のみ温度・湿度を計測
22   if (currState == measure) {
23     *tmp = readTmpSensor();
24     *hmd = readHmdSensor();
25   }
26 }
27 void output(
28   struct Tuple_Float_Float** data,
29   struct Tuple_Int_Int** transition
30 ) {
31   // measureの時のみデータを送信
32   if (currState == measure) send(data);
33
34   // 状態遷移に応じて周辺機器の電源モードを切替
35   switch ((*transition)->member1) {
36     case off2Sleep:
37       setupWiFi(); break;
38     case sleep2Off:
39       turnOffWifi(); break;
40     case sleep2Measure:
41       wifiDisableLowPowerMode();
42       setupTmpSensor(); setupHmdSensor();
43       break;
44     case measure2Off:
45       turnOffWifi();
46       turnOffTmpSensor(); turnOffHmdSensor();
47       break;
48     case measure2Sleep:
49       wifiEnableLowPowerMode();
50       turnOffTmpSensor(); turnOffHmdSensor();
51       break;
52   }
53   currState = (*transition)->member0;
54 }
```

図 4　センサーノードシステムのドライバーコード

辺機器のアクセスのタイミングの決定や周辺機器の電源管理に使用する．状態 `Off`,`Sleep`では，データ送信を行わないので `data`にダミーの値を設定している一方，状態 `Measure`では測定した `tmp`,`hmd`を設定している．状態遷移は `toggle`の値や `pulse1s`が立っている回数をカウントして秒数を数えるノード `t`に基づいて行う．ドライバーコード内の関数 `input`,`output`では，出力 `transition`の値に基づいて適切なタイミングで入出力を行う．`output`ではさらに周辺機器の電源モードの切替処理を出力された状態遷移に応じて行う．

この実装には問題点が複数ある．一つ目は，周辺機器の電源モード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ことである．そのため，状態遷移などシステムの仕様が変更された際にそれら二つのプログラム両方を修正する必要があり，メンテナンス性が低くなっている．また，ドライバーコード中の周辺機器の電源切替ロジックを誤って 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 プログラムには各周辺機器へのアクセスが現在有効かどうかが現れない．そのため，図 3，23-24 行目のように，電源が切れている機器の無効な値へのアクセスをコンパイル時に検出することができない．この問題点もシステムにバグや脆弱性を埋め込む余地を作っている．三つ目は，switch モジュールの次の状態や状態遷移を出力するコードがボイラープレートとなっていることである．switch モジュールの状態はモジュール内に閉じているため，状態に関する情報をモジュール外で使うには専用の出力ノードを用意する必要がある．また，switch モジュールのインスタンスがネストした階層的な状態遷移をもつシステムの場合には，状態遷移の情報を子インスタンスから親インスタンスへ引き渡していく必要がある．システムの状態遷移が複雑になるにつれ，ボイラプレートが増えていくことが予測できる．

4　提案手法

前節で述べた問題を解決するため，XStorm に対して**モード**という概念を導入した FRP 言語 XCios を提案する．モードの導入に際して構文要素及び実行モデルが変更されている．本節では，図 5 の XCios でのセンサーノードシステムの実装を通して，構文要素・実行モデルへの変更を説明する．

4.1　モード型

電源モードといった周辺機器がとる状態を言語内で表すため，**モード型**という enum のようなものを導入する．モード型は図 5，1-4 行目のように定義する．

```
# オンオフのみの電源モードに対応するモード型
mode OnOff = Off | accessible On
# Wi-Fiモジュールの電源モードに対応するモード型
mode WiFiMode = Off | LowPower | accessible On

switchmodule WiFiThermoHygrometer {
  in  toggle: Bool, pulse1s: Bool,
      tmp: 'OnOff Float, hmd: 'OnOff Float
  out data: 'WiFiMode (Float, Float)
  init Off

  state Off
    # 全ての周辺機器を電源オフ
    with tmp = Off, hmd = Off, data = Off
  {
    # アクセス可能でないのでコンパイルエラー
    # node invalid = tmp
    switch: if toggle then Sleep else Retain
  }

  state Sleep
    # Wi-Fiモジュールは低電力モード
    with tmp = Off, hmd = Off, data = LowPower
  {
    node t(0) = t@last + if pulse1s then 1 else 0
    switch:
      if !toggle then Off
      else if t >= 10 then Send
      else Retain
  }

  state Send
    # 全ての周辺機器を電源オン
    with tmp = On, hmd = On, data = On
  {
    node out data = (tmp, hmd)
    switch: if !toggle then Off else if Sleep
  }
}
```

図 5　XCios で記述されたセンサーノードシステム

定義はモード型の名前から始まり，続いてモード名が列挙される．**accessible**キーワードについてはこの項で後述する．モード型はモジュールの入出力ノードの型を修飾する形で使用する．例えば 9 行目では，出力ノード `data`は `(Float, Float)`型の値を出力とし `Off`, `LowPower`, `On`の 3 つのモードを持つ出力デバイスと対応していることを示している．なお，全ての入出力ノードをモード型で修飾する必要はないことに注意されたい．

同じモード型に属するモードには順序関係が定まっている．具体的にはモード型の定義時にモードが昇順に並んでいるとする．この順序関係の意図するところは「大きくなるほど消費電力が大きくなっていく」ことである．一般に消費電力が大きいほど周辺機器で使える機能が増えたり品質が良くなったりするので，この順序は「機能順」と対応しているとも言える．**accessible**キーワードは，それ以降に列挙されたモードにおいて周辺機器のデータの読み書きが有効であることを示す．**accessible**キーワードより前には電源オフやスリープモードといったモードが列挙され，**accessible**キーワード以降には電源オンや機能制限がかかったモードが列挙されることを想定している．図 5，4 行目の `WiFiMode`の例では，`Off < LowPower < On`であり，値を送信できるのは `On`の時のみであることを示している．他にも，例えばバックライト付きのディスプレイのモードは **mode** `DisplayMode = Off | ` **accessible** `NoBacklight | On`のように定義できる．この順序関係は，複数のインスタンスが同じ入出力ノードに対して異なるモードを要求した場合に利用される．詳細は第 4.5 項で後述する．

4.2　モード注釈

モード注釈は switch モジュールの各状態における入出力ノードのモードを指定する．モード注釈は，12-14 行目のように各 **state**ブロックの先頭で **with**キーワードに続けて各入出力のモードの指定を列挙する．各状態で異なるモード注釈を与えることにより動的な周辺機器の状態切替を宣言的に表現できるようになる．また，16-17 行目のように，モード注釈でアクセス可能でないと指定された入出力ノードの参照をコンパイル時にエラーとして検出できる．これにより第 3 節で示した無効なノードへアクセスできてしまう問題を解決している．

モード注釈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．周辺機器のモード切替は時間がかかることがあり，反応性向上のため意図的に周辺機器の電源をつけたままにしておきたい場合などがある．関係式から値が参照されないノードのモード注釈を大きいモードに指定しておくことで，それに対応する周辺機器のウォーミングアップを表現することができる．なお，アクセス可能であるモードが指定された出力ノードには必ず関係式が定義されてなければならない．これは関係式がないとドライバーコードに渡す出力ノードの値が不定となってしまうためで，XCios ではこのような状況をコンパイルエラーとする．

モード注釈は入出力ノードのモードを厳密に指定するものではなく，指定されたモード以上であること

```
1  void input_toggle(int* toggle) {
2    *toggle = readToggle();
3  }
4  void input_pulse1s(int* pulse1s) {
5    *pulse1s = readOscillator();
6  }
7  void input_tmp(float* tmp) {
8    *tmp = readTmpSensor();
9  }
10 void input_hmd(float* hmd) {
11   *hmd = readHmdSensor();
12 }
13 void output_data(
14   struct Tuple_Float_Float** data) {
15   send(data);
16 }
17 void hook_tmp_Off_to_On() {
18   setupTmpSensor();
19 }
20 void hook_tmp_On_to_Off() {
21   turnOffTmpSensor();
22 }
23 void hook_hmd_Off_to_On() {
24   setupHmdSensor();
25 }
26 void hook_hmd_On_to_Off() {
27   turnOffHmdSensor();
28 }
29 void hook_data_Off_to_LowPower() {
30   setupWiFi();
31 }
32 void hook_data_Off_to_On() {}
33 void hook_data_LowPower_to_Off() {
34   turnOffWifi();
35 }
36 void hook_data_LowPower_to_On() {
37   wifiDisableLowPowerMode();
38 }
39 void hook_data_On_to_Off() {
40   turnOffWifi();
41 }
42 void hook_data_On_to_Sleep() {
43   wifiEnableLowPowerMode();
44 }
```

図 6　XCios で記述したセンサーノードシステムのドライバーコード

を保証する．これについても，詳細は第 4.5 項で後述する．

4.3　フック

XCios は XStorm と同様に 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．フックは各入出力ノードのモードが切り替わった際にランタイムに自動で呼び出される関数である．図 6 に XCios で実装したセンサーノードシステムのドライバーコードを示す．例えば，17-19 行目の hook_tmp_Off_to_Onは入力ノード tmpのモードが Offから Onに切り替わる際に呼び出されるフックである．フック内に周辺機器の電源

```
1  {
2    // 任意の識別子
3    "multisensor": {
4      // どの入出力ノードと対応するか
5      "nodes": ["tmp", "hmd"],
6      // tmpとhmdのモードの組み合わせを
7      // multisensorのどのモードにマッピングするか
8      "mapping": {
9        // どちらかがOnの場合はmultisensorもOn
10       "On": [
11         ["On", "_"],
12         ["_", "On"]
13       ],
14       // 両方OffならmultisensorはOff
15       "Off": [["Off", "Off"]]
16     }
17   }
18 }
```

図 7　温湿度センサを使用する場合のモード割当て

```
1  void input_multisensor(float* tmp, float* hmd) {
2    readTmpAndHmd(tmp, hmd);
3  }
4  void hook_multisensor_Off_to_On {
5    setupMultisensor();
6  }
7  void hook_multisensor_On_to_Off {
8    turnOffMultisensor();
9  }
```

図 8　モード割当てによりまとめられたセンサーノードのコントローラのドライバーコード

モードの切替処理を実装すれば，ランタイムが適切なタイミングで自動的にその処理を呼び出してくれる．ユーザーは単に周辺機器それぞれのモード切替処理を記述するだけで良くなり，第 3 節で示したドライバーコードと比較してメンテナンス性が向上している．

4.4　モード割当て

センサーの中には，温湿度センサーのように複数項目を測定できるセンサー (複合センサー) がある．また，Wi-Fi モジュールのように入力と出力どちらも行う周辺機器もある．そのような，複数の入出力ノードに対応するような周辺機器の状態管理を行う際には，**モード割当て**を利用する．モード割当ては，プログラム内の入出力ノードのモードを実際に使用する周辺機器のどのモードに割り当てるかを設定するものである．モード割当てをコンパイル時に指定することで，それを考慮して input関数，output関数，フックがまとめられてコードが生成される．図 7 に，センサーノードシステムにおいて温湿度センサーを使用する場合のモード割当てを示す．現在は JSON ファイルで

パターンマッチのような形でモード割当てを記述することを検討している．例えば図 7 のモード割当ては，multisensorが入力tmp，hmdに対応し，そのどちらかのモードが Onなら multisensorも Onにすること表現している．このモード割当てを指定した際に生成されるコードのドライバーコードを図 8 に示す．モード割当てを指定しない場合と違い，図 6 では入力 tmp,hmdの入力処理やフックが一つにまとめられている．

4.5 実行時のモードの遷移

モードに関する機構を含めた実行モデルについて，まずベースとなる XStorm のものを示した上で説明する．

第 2 節でも述べたように，XStorm は各ノードの値を関係式に従って更新すること (イテレーション) を繰り返し行う実行モデルを持っている．ノードの更新順序はコンパイル時にノードの依存関係をもとに適切に定められる．なお，入力ノードの現在値はドライバコードの input関数により設定される．switch モジュールでは，現在の状態の **state**ブロックで定義されたノードの更新が終わった後は，**switch**節の式をもとに状態の更新計算を行う．

XCios 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入出力ノードのモードの更新およびフックの呼び出しを追加で行うようになっている．各イテレーションにおける入出力ノードのモードは実行時に決定される．一般に各入出力ノードのモードはモード注釈で指定されたものとは一致しない．なぜなら，異なる switch モジュールのインスタンスが同じ入出力に対して異なるモードを要求する場合があるからである．そこで，「大きいモードは小さいモードを兼ねる」ことを仮定し，各入出力ノードに現在要求されているモードの内最大のものを採るようにした．結果として，システム全体として必要最低限のモードが実際に周辺機器に要求されることになる．呼び出すフックは現在と直前のイテレーションにおけるモードにより判断される．フックの呼び出しは，複数のフックを並行に行うのではなく，一つずつ順番に同期的に行うことを考えている．

同じシステムを実装した際に，この実行モデルの変更による性能への影響はほぼないと考えられる．なぜなら，この実行モデルの変更は，ユーザーがドライバコード内に書いていたモード管理処理を単に代わりに行うようにしただけだからである．処理系の実装が済み次第評価実験を行いたい．

5 関連研究

Cios [12] は XCios の前身となる FRP 言語である．Cios ではモードをユーザーが定義することができず，電源のオンオフに対応するようなモードのみが言語に組み込まれている．しかし，大抵の周辺機器はスリープモードといったモードを他に持っている．そのため，そういった 3 つ以上のモードを持つ周辺機器を扱うにはドライバコードにロジックを書く必要があった．これは第 3 節で述べたのと同じ問題を引き起こす．また，Cios ではモード割当てが導入されておらず，複合センサーを扱うことができなかった．

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 Arrowised-FRP(AFRP) [4] という時変値ではなく時変値の関数 (シグナル関数) を組み合わせてプログラムを記述するパラダイムを持っている．Hailstorm には switch#という動的な振る舞いの切替を表現する組み込みの 2 引数のコンビネータが提供されている．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り，第二引数は第一引数の出力を入力としシグナル関数を出力する関数である．動的に振る舞いを切り替えるには，第二引数で引数に応じて違うシグナル関数を出力すれば良い．

```
switch# :
  SF r1 a b -> (b -> SF r2 b c) ->
  SF (r1 ∪ r2) a c
```

しかし，Hailstorm では条件に応じて違う入出力を行うことができない．Hailstorm はリソース型 [14] というリソースごとにつく固有の型を用いることで AFRP というパラダイムにおいて不整合なく入出力できることを保証している．switch#の第二引数では条件にかかわらず常に同じリソース型のシグナル関数を返さなければならない制約がある．そのため，動的に入出力の使用不使用が切り替わるようなシステムを表

現することができない．

Scade/Lustre [5] や Lucid [2] は同期的データフロー言語である．これらの言語では，FRP とは違い，ストリームと呼ばれる値の列を組み合わせることでリアクティブな振る舞いを記述する．これらの言語に対しても，状態遷移系を陽に記述する言語機構を導入した拡張が提案された [3]．本論文で提案した言語機構はこの拡張に適用することができると考えている．

Emfrp には文脈指向プログラミング (Context-Oriented Programming，COP) を導入する拡張も提案されている [13]．この拡張では，XStorm の状態遷移系を陽に記述する言語機構とは違う手法で動的な振る舞いの切替を表現する．具体的には，ある条件が満たされた時 (文脈) に上書きしたいノードの定義を層として定義し，実際に条件が満たされた層の定義を有効化する形で動的な振る舞いを表現する．しかし，周辺機器のモードを動的に切り替えることを表現する際には第 3 節で述べた XStorm と同様の問題が発生する．この COP 拡張においても，層ごとにモード注釈をつけられるような拡張を考えれば，周辺機器のモード制御をできるようになると考えている．

Statecharts [6] や Stateflow [7]，SyncCharts [1] は状態遷移のあるリアクティブシステムをグラフィカルに表現するツールである．これらのツールでは状態遷移の際に起こすアクションやイベントなどを表現できる．これは XCios におけるフックと対応する．しかし，我々の知る限りでは，状態に応じて使えるシンボルを制限するような機構はなく無効な値を周辺機器から読む表現ができてしまう．

6 まとめと今後の方針

小規模組込みシステム向け FRP 言語 XStorm の状態遷移モデルを発展させ，周辺機器のモードを制御するロジックの記述を可能にした FRP 言語 XCios を提案した．周辺機器のモードを表すためにモード型を導入し，switch モジュールの状態に各入出力のモードを指定するモード注釈を導入した．生成コードにはモード切替時にランタイムによって実行される関数フックを新たに導入し，モード割当てによって入出力関数やフックをマージできるようにした．

XCios の言語機構は FRP プログラム側が周辺機器のモードを制御することを可能にするものだが，周辺機器の中には自律的にモードを変えるものがある．そのような周辺機器を FRP プログラム側で扱うことについては将来課題とする．

また，XCios では入出力ノードを組み合わせて新たに仮想的な入出力を作るようなことができない．例えば，値としては二つの入力ノードの値の平均値を持ち，なおかつモードの切替が元の二つのノードに伝搬されるようなノードを考えられる．これを扱う言語機構についても今後検討していきたい．

今後は，XCios の処理系を実装しランタイムのオーバーヘッドやコードサイズについて評価する予定である．また意味論を形式化することも考えている．

謝辞 本研究の一部は JSPS 科研費 21K11822 および 22K11967 の助成を受けている．

参 考 文 献

[1] André, C.: Syncharts: A visual representation of reactive behaviors, 1995.
[2] Caspi, P. and Pouzet, M.: Synchronous Functional Programming : The Lucid Synchrone Experiment *, 2008.
[3] Colaço, J.-L., Pagano, B., and Pouzet, M.: A Conservative Extension of Synchronous Data-Flow with State Machines, *Proceedings of the 5th ACM International Conference on Embedded Software*, EMSOFT '05, New York, NY, USA, Association for Computing Machinery, 2005, pp. 173–182.
[4] Courtney, A. and Elliott, C.: Genuinely Functional User Interfaces, (2001).
[5] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D.: The synchronous data flow programming language LUSTRE, *Proceedings of the IEEE*, Vol. 79, No. 9(1991), pp. 1305–1320.
[6] Harel, D.: Statecharts: A Visual Formalism for Complex Systems, *Sci. Comput. Program.*, Vol. 8, No. 3(1987), pp. 231–274.
[7] MathWorks: Stateflow.
[8] 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, Mar. 2020.
[9] 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(2020), pp. 1–13.
[10] 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, 2020.

[11] 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, 2016, pp. 36–44.

[12] 瀧本哲史, 森口草介, 渡部卓雄: 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, 情報処理学会研究報告, Vol. 2022-EMB-60, No. 1(2022).

[13] 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, 2018, pp. 23–30.

[14] Winograd-Cort, D., Liu, H., and Hudak, P.: Virtualizing real-world objects in FRP, *Practical Aspects of Declarative Languages: 14th International Symposium, PADL 2012, Philadelphia, PA, USA, January 23-24, 2012. Proceedings 14*, Springer, 2012, pp. 227–241.