小規模組込みシステム向け FRP 言語における周期的タスクの記述方式

辻 裕太　森口 草介　渡部 卓雄

関数リアクティブプログラミング (FRP) は，時間とともに変化する値（時変値）によってプログラムを抽象化し，入力から出力への計算の宣言的な記述を可能にするプログラミングパラダイムである．本研究では，組込みシステム向け FRP 言語 Emfrp に対して周期的タスクの記述を可能にする構文および処理系の拡張を導入する．周期的なパルスあるいは時刻を時変値とすることで，Emfrp で周期的タスクを簡潔に表現することは可能であるが，従来の処理系で生成されるコードの実行効率は高くない．提案手法では，周期的タスクの実行におけるリアルタイム性の向上だけでなく，入力の取得頻度および各時変値の更新頻度を適切に選び，結果として実行効率の向上を可能にする．本稿では拡張した構文およびコンパイラによって生成されるコードについて述べ，これらについて例題を通して既存の Emfrp のものと比較し，本手法の有用性を示す．

1 はじめに

今日において，Web サイトやアプリのようなグラフィカルユーザーインターフェース (GUI) や家電，自動販売機, 券売機といった組込みシステムは, 我々の日常にありふれている．これらのような外部からの入力に対して応答を行うシステムはリアクティブシステムと呼ばれる．

リアクティブシステムを実装する方法は様々である．例えば，各イベントに対応したイベントハンドラを事前に登録しておき，その発生に応じて反応を行うイベント駆動型プログラミングの適用が挙げられる．また，ソフトウェア開発におけるデザインパターンの一つである Observer パターンを用いることで，プログラム内の状態の変更を検出して他のオブジェクトへ通知し，最終的に外部へ応答を行うといった手法も存在する．

関数リアクティブプログラミング (Functional Reactive Programming, FRP) は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の一つである．FRP では時間に依存して変化する**時変値 (time-varying value)** と呼ばれる値を扱い，時変値から別の時変値へ変換する計算を宣言的に記述することで，時事刻々と変化する入力と出力の関係性を見通しよく表現することができる．FRP は Web ベースのアプリケーション開発 [2]，ゲーム制作 [1]，ロボット制御 [4] といった典型的なリアクティブシステムに対して適用されている．

Emfrp [6] は小規模組込みシステム向け FRP 言語である．この言語の特徴として，記述したプログラムの実行時におけるメモリ使用量の上限の推定が可能である点や，再帰的データ構造や再帰関数の禁止およびループの不使用によって時変値計算の停止性が保証されているという点が挙げられる．ところで，組込みシステムでは定期的なセンサー値の取得や一定のフレームレートで行うディスプレイへの描画といった周期的タスクの実行がしばしば求められる．Emfrp プログラムは言語処理系によって C++プログラムへとコンパイルされるが，このような周期的タスクを効率的に実行するランタイムを生成することができないという課題がある．

そこで本研究は， Emfrp に対する拡張 PbEmfrp

A Method for Describing Periodic Tasks in the FRP Language for Small-Scale Embedded Systems.
Yuta Tsuji, Sosuke Moriguchi, Takuo Watanabe, 東京工業大学, Tokyo Institute of Technology.

を提案する．PbEmfrp では Emfrp の時変値に対する時間的特徴付けを行い，その情報を用いて時変値計算を行うコードを生成する．これにより，周期的タスクを記述した際のリアルタイム性の向上だけでなく，計算資源を浪費しない，より効率的な実行を実現する．

本論文は，次のように構成される．まず第 2 章にて，研究の動機とともに本研究が拡張の対象とする既存の FRP 言語である Emfrp について述べる．次に第 3 章にて，提案手法である PbEmfrp の構文や新たに導入した各種概念について説明し，そのコード生成の実装方針を第 4 章で示す．ケーススタディを通して PbEmfrp の具体的な適用例を第 5 章で提示し，そして第 6 章では関連研究を与え，最後に第 7 章で結論と今後の課題について述べる．

2 研究の動機

Emfrp は組込みシステム向け FRP 言語である．Emfrp プログラムには，各時変値の関係を式（以下，更新式）として記述することで入力時変値から出力時変値を得る一連の計算を表現する．プログラムは入力に対応する値を取得した後，更新式にしたがって各時変値の値を得る．全ての時変値の値を取得した後，出力時変値を外部へ送出する．入力から出力への一連の処理は **イテレーション (iteration)** と呼ばれ，ランタイムによって繰り返し実行される．

Emfrp コード 1 は，温度と湿度から不快指数を計算し，その値によってファンの回転数の制御を行うものである．ここではヒステリシス制御が利用されており，スイッチが不快指数の閾値付近で ON/OFF を頻繁に繰り返すのを防いでいる．

ところで，ある 2 つの時変値 u, v の依存関係に循環が存在する場合，任意のイテレーションにおいて u を計算するのに v の現在の値が必要となり，v を計算するために u の現在の値が必要となる．この場合は時変値の更新順序が定まらない．そのため，コンパイル時に全ての時変値の更新順序を決定する Emfrp コンパイラは，そのようなプログラムに対してコンパイルエラーを生成する．そこで Emfrp には @last というオペレータが定義されており，対象となる時変値に

```
1  module SimFanController  # モジュール名
2  in  tmp : Double,        # 温度センサーの値
3      hmd : Double         # 湿度センサーの値
4  out fan : Bool
5      # ファンのスイッチの状態を示す論理値
6  use Std
7
8  # 不快指数
9  node di = 0.81 *. tmp +. 0.01 *. hmd *.
10           (0.99 *. tmp -. 14.3) +. 46.3
11
12 # ファンのスイッチの状態
13 node init[False] fan = di >=. th
14
15 # ヒステリシス制御に用いる閾値
16 node th = 75.0 +.
17        (if fan@last then -.0.5 else 0.5)
```

ソースコード 1　SimFanController.mfrp

おける直前のイテレーションで計算された値を参照することができる．これにより時変値間の依存関係を非循環なものに変えることができ，コンパイラは時変値の更新順序を決定できるようになる．このプログラムでも `fan` と `th` という 2 つの時変値が互いに参照し合っているが，`th` はその定義において `fan@last` で `fan` を参照しており，時変値間の依存関係の循環を回避している．

Emfrp は，一定の時間間隔でシステムに対する入力から出力を生成する周期的タスクを記述することも可能であるが，そこにはいくつかの課題が存在する．ここでは周期的タスクの例としてデジタル温湿度計を取り上げ，その実装を示す．この温湿度計は周期的に温度および湿度を取得し，これらから計算される不快指数とともにディスプレイに表示するだけでなく，外部記録装置に対して過去のセンサー値を用いた移動平均を記録として送信する機能を持つ．以下は温湿度計の仕様を示したものであり，図 1 は入力から出力を得る一連の計算の流れを示したものである．

- 入力
  - 温度センサーの値
    - 5000ms 後から 5000ms 間隔で値を取得する
  - 湿度センサーの値
    - 5000ms 後から 3000ms 間隔で値を取得する

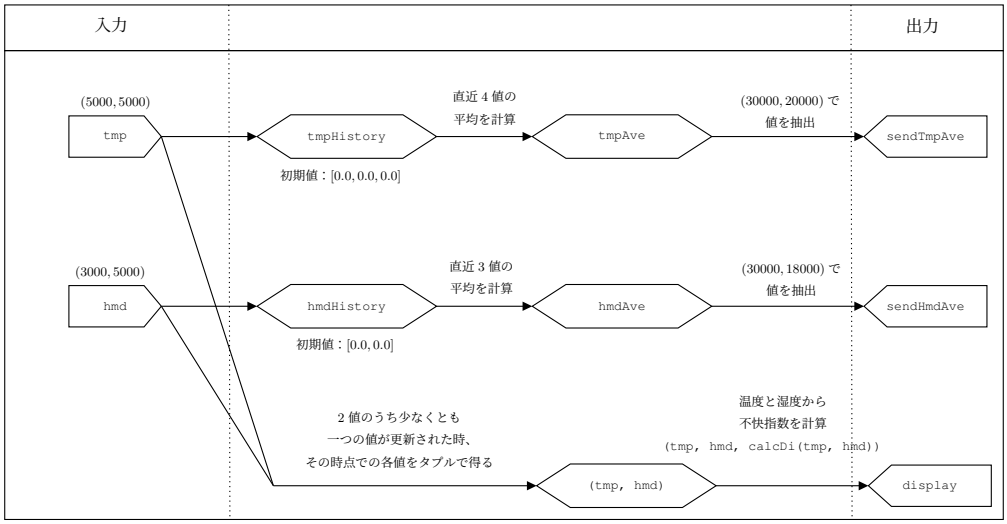
- 出力
  - 外部の記録機器へ送信する，温度の 4 値の加重平均
    - 20000ms 後から 30000ms 間隔で送信する
  - 外部の記録機器へ送信する，湿度の 3 値の加重平均
    - 18000ms 後から 30000ms 間隔で送信する
  - ディスプレイに表示する，温度，湿度，不快指数の組
    - 温度か湿度をセンサーが読み取ったタイミングで表示する

ソースコード 2 は温湿度計の Emfrp による実装である．

```
module ThermoHygrometer
in  tmp : Double, # 温度
    hmd : Double, # 湿度
    time: Int  # プログラム開始からのミリ秒
out sendTmpAve: Double,
    # 温度の移動平均
    outputSendTmpAve: Bool,
    # 温度の出力を行うかを示す論理値
    sendHmdAve: Double,
    # 湿度の移動平均
    outputSendHmdAve: Bool,
    # 湿度の出力を行うかを示す論理値
    display   : (Double, Double, Double),
    # 温度，湿度，不快指数の組
    outputDisplay: Bool
    # ディスプレイへ出力を行うかを示す論理値
use Std

# 温度の移動平均の計算
node init[(0.0, 0.0, 0.0, 0.0)] tmpHistory =
  if tmpUpdated
    then tmpHistory@last of
      (tmp1, tmp2, tmp3, tmp4) -> (tmp, tmp1, tmp2, tmp3)
    else tmpHistory@last

node sendTmpAve = tmpHistory of
  (tmp1, tmp2, tmp3, tmp4) -> (tmp1 +. tmp2 +. tmp3 +. tmp4) /. 4.0

# 経過時刻をもとに，出力を行うか決定
node init[20000] tmpAveTargetTiming =
  tmpAveTargetTiming@last + (
    if outputSendTmpAve then 30000 else 0)
node outputSendTmpAve =
  time > tmpAveTargetTiming@last


# 湿度の移動平均の計算
node init[(0.0, 0.0, 0.0)] hmdHistory =
  if hmdUpdated
    then hmdHistory@last of
      (hmd1, hmd2, hmd3) -> (hmd, hmd1, hmd2)
    else hmdHistory@last

# 出力時変値
node sendHmdAve = hmdHistory of
  (hmd1, hmd2, hmd3) -> (hmd1 +. hmd2 +. hmd3) /. 3.0

# 経過時刻をもとに，出力を行うか決定
node init[30000] hmdAveTargetTiming =
  hmdAveTargetTiming@last + (
    if outputSendHmdAve then 18000 else 0)
node outputSendHmdAve =
  time > hmdAveTargetTiming@last

# 描画する情報の計算
func calcDi(tmp, hmd) =  # 不快指数
  0.81 *. tmp +. 0.01 *. hmd
  *. (0.99 *. tmp -. 14.3) +. 46.3

node display = {
  (tmp1, tmp2, tmp3, tmp4) = tmpHistory
  (hmd1, hmd2, hmd3) = hmdHistory
  (tmp1, hmd1, calcDi(tmp1, hmd1))
}

# 経過時刻をもとに，出力を行うか決定
node init[5000] tmpTargetTiming =
  tmpTargetTiming@last + (
    if tmpUpdated then 5000 else 0)
node tmpUpdated =
  time > tmpTargetTiming@last

node init[5000] hmdTargetTiming =
  hmdTargetTiming@last + (
    if hmdUpdated then 3000 else 0)
node hmdUpdated =
  time > hmdTargetTiming@last

node outputDisplay =
  tmpUpdated || hmdUpdated
```

ソースコード 2　ThermoHygrometer.mfrp

Emfrp プログラムはその実行においてイテレーションが繰り返し行われ，各イテレーションの実行時間は一定ではない．したがって温湿度計の仕様に示されるようなタスクの周期的実行を Emfrp プログラムに記述するには，例えば入力時変値としてプログラムが開始してからの経過時間を取り，各出力時変値に対してその実行タイミングを表現するパルスを生成し，タスクを実行すべき時刻を処理系が生成する C++プログラム上で検出する必要がある．この周期的タスクの実装方法は，いくつかの欠点を抱えている．まず，プログラマが時刻を用いてパルスを生成するロジックを逐一記述する必要がある点が挙げられる．仮に



(π, φ)：実行周期が π[ms] であり、最初の実行が行われる、プログラム開始からの
経過時刻が φ[ms] であるようなタイミングで実行されることを示す

図 1　温湿度計の構成

この処理がライブラリとして提供された場合においても，周期的タスクを実行するために必要な時刻情報を出力時変値として用意する必要がある．次に，周期的タスクの実行タイミングが前後する，ジッターと呼ばれる現象が起こる点が挙げられる．これは各イテレーションの実行開始時間の間隔が一定ではなく，時変値計算の開始時刻に周期性が無いという性質に起因している．最後に，計算資源の浪費が著しい点が挙げられる．Emfrp では各イテレーションにおいて必ず入力時変値から出力時変値を求める計算が行われ，より低頻度の入力および出力が要求されている状況においても，不要な計算を省略することができない．以上の議論から，Emfrp プログラムにおいて，計算資源を浪費しないような周期的タスクの記述は困難であると言える．

そこで本研究では，Emfrp の拡張として PbEmfrp を提案し，周期的タスクの簡潔な記述，ジッターの無い周期的タスクの実行，計算資源を浪費しないコードの生成を試みる．第 3 章では構文および言語機能に関する拡張についてを述べていき，第 4 章にてこの拡張を行なった PbEmfrp プログラムに対するコード生成の実装方針を述べていく．

3　PbEmfrp

PbEmfrp(Period-based Emfrp) は，周期的タスクを簡潔に記述し，かつ効率的なコード生成が行えるよう作成された，Emfrp のサブセットに対する拡張を行なった言語である．PbEmfrp における言語拡張は以下の通りである．

- 時変値への時間的特徴付け
- タイミング注釈
- シーケンス及び演算の導入
- 構文の拡張
- 全時変値に対するシーケンスの静的検査

以降，これらについて各々述べていく．

3.1　時変値の時間的特徴付け

Emfrp では，イテレーション内で全ての時変値の更新式を計算していたが，PbEmfrp ではイテレーションの概念がなく，それぞれの時変値に定められた **タイミング (timing)** において更新式を計算する．例えば，時変値 v のタイミングを，単位がミリ秒であるような周期と最初の開始時刻の組 (π, φ) によって表す．つまり，プログラムが開始してからの経過時刻 φ において最初に時変値 v の計算が行われ，以後 π 毎に

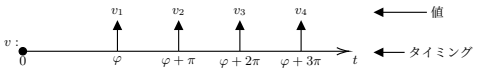


図 2　時変値の発生の模式図

(つまり時刻 $\varphi+\pi, \varphi+2\pi, \ldots$ に) 再計算が行われる．この計算が行われることを v の **発生 (occurrence)** と呼ぶ．図 2 は v の発生を表す図式である．一般に周期的タスクの実行時刻はその実行周期と実行開始時刻を用いて記述できることから，(複数の) 時変値のタイミングを対応づけることで周期的タスクの実行を表現できる．

Emfrp においては，プログラムに記述された時変値計算から得られる依存関係に基づいて全ての時変値が順次計算され，このイテレーションが幾度となく繰り返されるため，常に時変値計算が発生する．したがって，組み込みシステムにおいてはセンサーやモジュールからの値の読み込みおよび外部デバイスへの出力もそれに応じて頻繁に発生することになる．しかし PbEmfrp では，全ての時変値の全ての発生時刻がコンパイル時に決定されており，発生時刻以外では時変値計算を行わないようなコードが生成されるため，計算資源を浪費せず，余分な入力値の取得や出力の生成をしないような周期的タスクの実行を実現できる．

3.2　シーケンス

PbEmfrp における時変値は，その時間的特徴に応じて以下の 3 種類に分類される．

- 離散時変値
- 連続時変値
- 定時変値

離散時変値は PbEmfrp において中心となる概念であり，その発生時刻を示すタイミングを持つ．離散時変値には 0 個以上の初期値を持たせることができる．ある離散時変値について，それが持つ初期値の数をその**初期値数**と呼ぶ．後述する演算によって過去の発生における値にアクセスする際，まだ発生が存在していない場合の値として初期値が用いられる．例えばある離散時変値の初期値数が l であるとき，l 回前までの発生における値は常にアクセスできるが，それよりも過去の発生における値にはアクセスできない．連続時変値は離散時変値とは異なり，時変値がいかなる時刻においても値を持つ．連続的な値は本質的に直接コンピュータ上で計算することができないため，この時変値は離散時変値としてサンプリングされて利用される．定時変値は連続時変値と同様に連続的に値を取る時変値であるが，常に一定の値を取る．この特徴から，他の時変値との計算を簡略化できるため，連続時変値とは区別される．

PbEmfrp では時変値の時間的特徴を区別するために**シーケンス (Sequence)** という概念を用いる．ここでは便宜上，発生を持たないことを示すシーケンスを加えて以下の 4 つの値のいずれかをとる．

(θ, l) :　離散時変値のタイミングθと初期値数 l の組
'cont :　連続時変値であることを示すシーケンス
'const :　定時変値であることを示すシーケンス
'never :　発生を持たないことを示すシーケンス

ところで，離散時変値はその発生時にのみ値を持つと解釈される．しかし，このままでは異なるタイミングを持つ離散時変値を参照すると値を持たない場合がある．この問題を解決するために，PbEmfrp では時変値から別のシーケンスを持つ時変値を生成する演算が定義されている．

- サンプリング
- 時変値合成
- @last 参照
- @now 参照

3.2.1　サンプリング

サンプリングは e `@>` θ で表現され，あるタイミングを持つ離散時変値 e に対して，部分的な発生を表すタイミング θ における e の値を取得するような離散時変値へと変換する．図 3 は，サンプリングの意味を表現したものである．

3.2.2　時変値合成

時変値合成は左結合の三項演算子 e_1 `@<` f `>@` e_2 で表現され，図 4 は生成される離散時変値を示したものである．合成された時変値は，一方の離散時変値が発生し他方がそうでない場合，発生した方の値をそのまま返す．各時変値の発生時刻が重複していた場合，

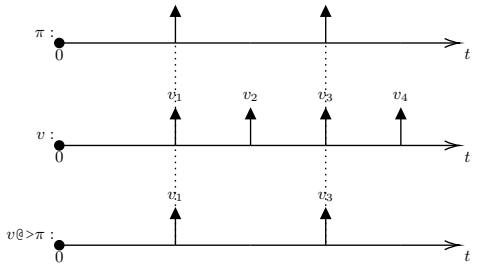


図 3　時変値のサンプリング

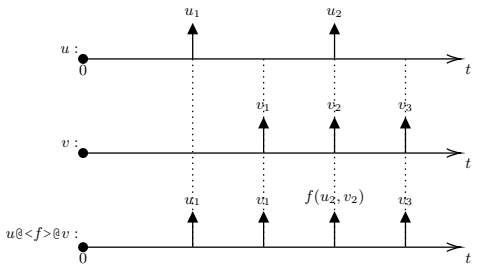


図 4　時変値合成

与えられた関数 f に各時変値の値を適用し，その結果を返す．つまり，与えられる関数 f は二引数関数であり，また各入力の型と出力の型は一致している必要がある．

この演算の結果のタイミングは，少なくとも一方の時変値が発生しているようなタイミングに対応する．これを**タイミングの合成**と呼び，各離散時変値のタイミングが θ_1, θ_2 であるとき，$\theta_1 \mid \theta_2$ で表される．

3.2.3　@last 参照

@last 参照は v`@last`$[n]$ として表現され，図 5 は生成される離散時変値を示したものである．この演算は，初期値が存在する時変値に対して n 回前に計算された値を持つ離散時変値を生成する．また v の時変値計算が行われていない場合における@last 参照の値は，v の定義で記述した初期値に対応する．

なお，v`@last`$[n]$ が生成する時変値は，v のシーケンスにおける初期値数 l の値が n だけ減少したようなシーケンスを持つ．残った初期値数は，他の時変値を定義する際の初期値列を生成するために用いられる．

3.2.4　@now 参照

初期値が存在する，すなわち初期値数 l が正であるような離散時変値 v に対して，@now 参照は連続時

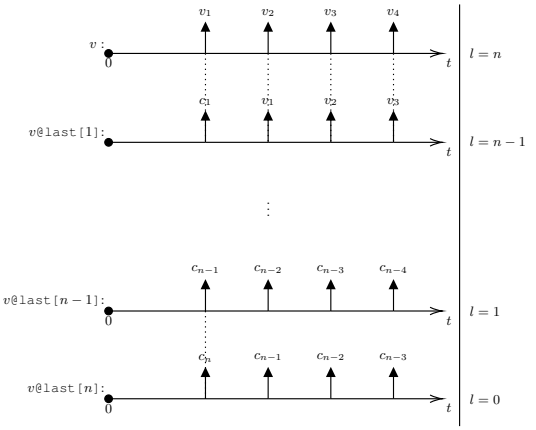


図 5　@last 参照

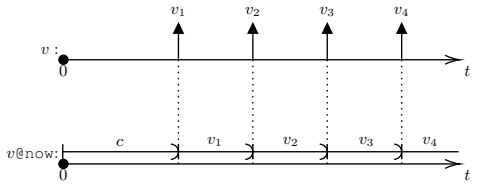


図 6　@now 参照

変値を生成する．最初の発生までは v の初期値をとり，その後は各々の発生の値をその次の発生までとる．図式では v`@now` は図 6 のように表現する．

3.2.5　多項演算

多項演算においては，一般に異なるシーケンスを持つ複数の時変値に対する演算を行う．しかし前述の通り，離散時変値のタイミングが異なる場合については値を同時に取り出すことができないため，サンプリングを明示的に行うことでこれを解決する必要がある．それ以外の場合については，以下の議論のように時変値の種類に応じて計算結果のシーケンスを決定する．まず，タイミングが等しい離散時変値を扱う場合および連続時変値や定時変値と離散時変値が混合している場合については，離散時変値の初期値数を考慮しつつ適宜サンプリングを行う必要がある．異なる 2 つの離散時変値 v_1, v_2 の初期値数を各々 l_1, l_2 とすると，演算結果の時変値が持つ初期値は v_1, v_2 が持つ各初期値を用いて計算される．しかし l_1, l_2 が異なる場合，双方から同時に初期値を取り出すことができるのは $\min(l_1, l_2)$ 回前の発生までである．したがって，

v_1 と v_2 の計算結果は初期値を $\min(l_1, l_2)$ 個持つ．

次に，一方が離散時変値 v であり他方が連続時変値である場合，連続時変値は初期値を持たないことから演算結果も初期値を持たず，初期値数は 0 となる．定時変値は連続時変値と同様に連続的に値を取りうるが，いかなる時刻においても一定値 c をとる．この性質から，どのようなタイミングでサンプリングしても，初期値が c であり全ての発生で c を返すような離散時変値となる．したがって演算結果の初期値は v の初期値と c によって計算できるため，得られる時変値の初期値数は v と同じである．

最後に，双方が離散時変値でない場合も考慮する必要がある．この場合の演算結果のシーケンスは，定時変値間の演算のみ 'const となり，そうでない場合は 'cont となる．これは，少なくとも一方が連続時変値である場合は演算結果の値が必ずしも一定ではなくなるためである．

以上の議論より，2 つの時変値に関する演算から得られる時変値のシーケンスは表 1 の通りである．ただし 3 つ以上の時変値に関する演算については，以上のシーケンスの検査を最初の 2 項から順次適用する．

v_1 \ v_2	(θ, l_2)	'cont	'const	'never
(θ, l_1)	$(\theta, \min(l_1, l_2))$ *	$(\theta, 0)$	(θ, l_1)	'never
'cont	$(\theta, 0)$	'cont	'cont	'never
'const	(θ, l_2)	'cont	'const	'never
'never	'never	'never	'never	'never

* タイミングが一致しない場合，'never を返す．

表 1　v_1 と v_2 の演算におけるシーケンス

3.3　タイミング注釈

タイミング注釈は，プログラマが入出力時変値の宣言や離散時変値のサンプリングを行う際に用いる注釈であり，以下の 3 種類が存在する．

- 定タイミング
- 時変値へのタイミング参照
- 匿名タイミング

定タイミングは時変値のタイミングを直接与える方法であり，周期的タスクの実行周期と実行開始時刻の組で記述する．また時変値へのタイミング参照は，例えば同じタイミングを持つ入出力時変値に対するタイミング注釈の記述を簡潔にしたり，異なるタイミングを持つ 2 つの離散時変値について，サンプリング演算を用いて一方に他方のタイミングを合わせる処理を記述する際に有用である．そして匿名タイミングは，シーケンスの検査の過程で時変値のタイミングを推定させる用途で記述する．これは例えば，入力時変値を定タイミングで与え，時変値合成によって出力時変値のタイミングが複雑になった場合，このタイミングを推定させる目的で用いることができる．PbEmfrp の言語処理系はこれらの注釈から全ての時変値のシーケンスを決定し，得られたシーケンスの値を用いてコード生成を行う．

3.4　構文

簡単のため，PbEmfrp は構文のベースとして Emfrp のサブセットを用い，これに対して拡張を行う．形式的な構文定義および使用するメタ変数は，それぞれ図 9 および図 7 を参照されたい．

3.4.1　モジュール

PbEmfrp のモジュールは，以下の 4 要素からなる．

- 入力時変値宣言
- 出力時変値宣言
- 時変値定義
- 関数定義

入出力時変値宣言では，PbEmfrp プログラムに対して入力および出力を行うタイミングをタイミング注釈として記述する．時変値定義は Emfrp と異なり，初期値を複数設定することができる．

3.5　シーケンスの静的検査

PbEmfrp プログラムにおける全ての時変値は各々シーケンスが定まっている必要があるが，入出力時変値を除いて，プログラマはこれを明示的に記述しない．これは，入出力時変値やサンプリング演算に与えられたシーケンス注釈をもとに PbEmfrp の処理系がシーケンスの静的検査を行うためである．そして全ての時変値に対してシーケンスが決定した場合にのみコードを生成し，そうでない場合はコンパイルエラー

v : 時変値名
τ : データ型
π : 実行周期
φ : 初期オフセット
l : 離散時変値の初期数
f : 関数
n : 整数
c : 定数
q : タイミング注釈

図 7 PbEmfrp の各種定義で用いるメタ変数

を生成する．PbEmfrp の各構文要素に対する形式的な型判断 (type judgements) と変数はそれぞれ図 11, 10 の通りであり，そこで利用される各種演算は図 8 の通りである．

4 コード生成

4.1 時変値の表現方法

Emfrp では，全ての時変値に対して現在のイテレーションの値と直前のイテレーションの値を長さ 2 の配列として持たせ，各イテーションでアクセスするインデックスを 0 と 1 で切り替えることでプログラムを実行していた．これに対して PbEmfrp では，時変値 v が持つシーケンスによって必要なメモリ領域のサイズが異なる．まず v が連続時変値の場合，任意のタイミングでサンプリングされる可能性がある．したがって必要なタイミングで値を取り出すことができるよう，関数として生成する．次に v が定時変値の場合，これはいかなる時刻においても一定の値を生成するため，コンパイル時に計算を行うことができる．そのため，これは計算結果を示す定数として展開される．最後に v が離散時変値の場合，必要なメモリサイズ m_v は以下で計算される．

$$m_v = \mathrm{size}(\tau_v) * (l_v + 1) + \mathrm{size}(\tau_{\mathrm{Int}})$$

ただし，

$\mathrm{size}(\tau)$: 型τのサイズ
τ_v : v の型
Int : 整数型
l_v : v の初期値数

このように得られるのは，次の理由からである．まず，初期値数が l である時変値 v は，その l 回前の発生の値を参照するために $l+1$ 要素の配列 a に履歴を記録しておく必要がある．その記録の更新方法についてであるが，単純には $i(\in 1, \ldots, l)$ 回前の発生の値を $a[i]$ に保持するために，発生の値を計算する前に $a[0]$ から $a[l-1]$ までの値を $a[1]$ から $a[l]$ までコピーすることで実現できる．しかしこれは v のサイズが大きい際にコピーコストが高くなってしまうため，最新の値を格納するインデックスおよび過去の発生の値を参照する際のインデックスをずらすことでコピーを回避する．つまり，i 回前の発生の値を示す初期値を $a[i-1]$ へ代入することで初期化し，最新の値を格納するインデックス j を l とする．以降は，各発生で v`@last`$[i]$ の参照に対しては $a[j+i \pmod{l+1}]$ を返すようにして更新式を計算し，$a[j]$ に格納，その発生での処理が終わるとインデックスを $j+l \pmod{l+1}$[†1] に更新する．このように実装することで，インデックス j のサイズは増えるもののアクセスするインデックスを変更するのみで履歴のコピーは不要になり，全体として必要なメモリ使用量は v の型のサイズの $l+1$ 倍と，インデックス j のサイズの和となる．

4.2 時変値の更新順序

PbEmfrp における時変値更新は，発生した時変値のインデックスの更新と現在値の更新からなる．インデックスを更新した直後は，現在値への参照が履歴の中で最も古い発生における値を指すため，時変値の更新を無秩序に行うと，プログラムに記述した通りの計算が行われなくなる可能性がある．そこで，過去の発生の値を参照しないような更新順序を決定する必要がある．

この問題は，Emfrp の言語処理系が行う時変値の更新順序の決定方法と同様のアプローチをとることで解決できる．まず @last 参照を除く時変値間の参照における全ての依存関係を抽出し，これらに対してトポロジカルソートを適用する．すると，循環した時変

†1 直感的には $j-1$ に対する剰余だが，負になる場合を考慮して $l+1$ を加えている．

$$
\mathrm{minify}(s_1, s_2) \quad = \quad \begin{cases} s_1 & (s_2 = \text{'const}) \\ s_2 & (s_1 = \text{'const}) \\ (\theta_1, 0) & (s_1 = (\theta_1, l_1) \wedge s_2 = \text{'cont}) \\ (\theta_2, 0) & (s_2 = \text{'cont} \wedge s_2 = (\theta_2, l_2)) \\ \text{'cont} & (s_1 = s_2 = \text{'cont}) \\ (\theta, \min(l_1, l_2)) & (s_1 = (\theta, l_1) \wedge s_2 = (\theta, l_2)) \\ \text{'never} & (\text{otherwise}) \end{cases}
$$

$\theta_1 \leq \theta_2$: θ_2の任意の発生時刻において,θ_1が発生している

$\theta_1 \mid \theta_2$: タイミングの合成

$\langle v, e \rangle \leftarrow vs$: $\texttt{node}\, v = e \in vs$のとき,$v, e$ の組を得る操作

$\langle v, n, e \rangle \leftarrow vs$: $\texttt{node}\, \texttt{init}[c_1, \ldots, c_n]\, v = e \in vs$ のとき,v, n, e の組を得る操作

$\langle f, n \rangle \leftarrow fs$: $\texttt{func}\, f(v_1, \ldots, v_n) = e \in fs$ のとき,f, n の組を得る操作

図 8　型判断に用いる演算

宣言

M	::=	(I, O, L, fs)	(モジュール)
I	::=	$\texttt{in}\, v_1 : \tau_1\, q_1, \ldots, v_n : \tau_n\, q_n$	(入時変値宣言)
O	::=	$\texttt{out}\, v_1 : \tau_1\, q_1, \ldots, v_n : \tau_n\, q_n$	(出力時変値宣言)
L	::=	$vdef^*$	(時変値定義)
$vdef$	::=	$\texttt{node}\, v = e$	(初期値無し時変値定義)
	|	$\texttt{node}\, \texttt{init}[c_1, \ldots, c_n]\, v = e$	(初期値付き時変値定義)
fs	::=	$(\texttt{func}\, f(v_1, \ldots, v_n) = e)^*$	(関数定義)

タイミング注釈

q	::=	$\text{'}(\pi, \varphi)$	(定タイミング)
	|	$\text{'}v$	(時変値へのタイミング参照)
		$\text{'}\_$	(匿名タイミング)

式

e	::=	c	(定数)
	|	v	(時変値参照)
	|	$(e_1, \ldots, e_n)$	(タプル)
	|	$\mathrm{uop}(e)$	(単項演算)
	|	$\mathrm{bop}(e_1, e_2)$	(二項演算)
	|	$\texttt{if}\, e_1\, \texttt{then}\, e_2\, \texttt{else}\, e_3$	(条件式)
	|	$f(e_1, \ldots, e_n)$	(関数適用)
	|	$e\, \texttt{@>}\, q$	(サンプリング)
	|	$e_1\, \texttt{@<}\, f\, \texttt{>@}\, e_2$	(時変値合成)
	|	$v\texttt{@last}[n]$	(@last 参照)
	|	$v\texttt{@now}$	(@now 参照)

図 9　PbEmfrp の構文

環境

$$
\begin{array}{lcll}
\Sigma & ::= & \emptyset \mid \Sigma\,[v \mapsto s] & \text{(時変値環境)} \\
fs & ::= & \emptyset \mid fs\,[\texttt{func}\, f(v_1, \ldots, v_n) = e] & \text{(関数環境)} \\
vs & ::= & \emptyset \mid vs\,[\texttt{node}\, v = e] \mid vs\,[\texttt{node}\, \texttt{init}[c_1, \ldots, c_n]\, v = e] & \text{(時変値定義環境)}
\end{array}
$$

変数

$$
\begin{array}{lcll}
\theta & ::= & (\pi, \varphi) & \text{(タイミング)} \\
 & \mid & (\pi, \varphi) \mid \theta & \text{(タイミングの合成)} \\
s & ::= & (\theta, l) & \text{(離散時変値)} \\
 & \mid & \text{'cont} & \text{(連続時変値)} \\
 & \mid & \text{'const} & \text{(定時変値)} \\
 & \mid & \text{'never} & \text{(発生がないシーケンス)}
\end{array}
$$

図 10　シーケンスの型判断に用いる変数

値間の依存関係が記述されていない限りは有向非循環グラフ (DAG) が得られる．この DAG はある時変値の計算にどの時変値の現在値が必要かを示したものになっているため，DAG の根にあたる依存先を持たない時変値から順に時変値更新を行なっていけば，求めたい計算順序が得られる．

4.3　周期的タスクの実行方法

周期的タスクは一度タスクの処理が完了すると次にタスクを実行する時刻まで待機するが，この際にビジーループを用いると計算資源を浪費してしまう．そこで PbEmfrp では，記述された全ての時変値の発生に応答できる最長の実行周期 T を言語処理系によって決定し，その周期を持つタイマー割り込みの設定を行うことでこの問題に対処する．

ところで，Emfrp プログラムでは 1 イテレーションを行う処理のみを記述し，それ自体には副作用を持たない．外部デバイスからの読み取りや書き込みのようなハードウェア依存となり得る処理は，Emfrp の言語処理系が提供する中身が空の関数スケルトン内に記述する．つまり Emfrp プログラム自体のコンパイルはハードウェアに依存せず，関数スケルトンを変更するだけで同じイテレーションを行う実行可能バイナリを生成することができる．PbEmfrp はタイマー割り込みを利用するが，このハードウェア依存の処理も Emfrp 処理系のコード生成と同様に，プログラマがタイマー割り込みの各種設定を行うことができるようなコード生成を行う．このときタイマー割り込みで実行するハンドラを設定する必要があるため，割り込み時に行う処理が書かれた関数が参照できるようになっている．

Emfrp では入力（出力）を行うタイミングが同期されており，全ての入力（出力）は同時に処理される．そのため，関数スケルトンはプログラムの入力（出力）に対し 1 つしか提供されない．一方 PbEmfrp では入出力を行うタイミングが同期されていないため，各時変値に対してそれぞれ独立に入出力を設定する必要がある．したがって，入力時変値と出力時変値の個数だけ関数スケルトンを生成する．

最後に，PbEmfrp が提供するプログラムでは各種の初期化後に割り込みを待機するための無限ループを行う．このループも関数スケルトンとして提供されており，ここに記述することで待機時にハードウェア依存の割り込み待ちを利用できる．ハードウェアによってはタイマー割り込みを低消費電力状態で待機する機能が備わっており，例えば ARM アーキテクチャには割り込み命令を受け取るまで CPU を低消費電力状態へと移行する WFI (Wait for Interrupt) 命令が存在している．これをループ内で用いて各タイマー割り込みを待機することにより，ビジーループを行う場合と比較して消費電力を抑えることができる．

以上をまとめると，PbEmfrp の処理系がプログラ

タイミング注釈

$$\vdash_q \text{'}(\pi,\varphi)\colon (\pi,\varphi) \quad \text{(S-CONSTPERIOD)} \qquad \frac{fs, \Sigma \vdash_e v\colon (\theta, l)}{fs, \Sigma \vdash_q \text{'}v\colon \theta} \quad \text{(S-VARPERIOD)}$$

$$fs, \Sigma \vdash_q \text{'\_}\colon \theta \quad \text{(S-ANONYMOUSPERIOD)}$$

式

$$\vdash_e c\colon \text{'const} \quad \text{(S-CONST)}$$

$$\frac{fs, \Sigma \vdash_e v\colon (\theta, l) \qquad 0 < n \leq l}{fs, \Sigma \vdash_e v\texttt{@last}[n]\colon (\theta, l-n)} \quad \text{(S-LAST)} \qquad \frac{fs, \Sigma \vdash_e v\colon (\theta, l) \qquad l > 0}{fs, \Sigma \vdash_e v\texttt{@now}\colon \text{'cont}} \quad \text{(S-NOW)}$$

$$\frac{\langle v\colon s\rangle \in \Sigma}{fs, \Sigma \vdash_e v\colon s} \quad \text{(S-VAR)} \qquad \frac{fs, \Sigma \vdash_e e\colon s}{fs, \Sigma \vdash_e \mathrm{uop}(e)\colon s} \quad \text{(S-UOP)}$$

$$\frac{fs, \Sigma \vdash_e e_1\colon s_1 \qquad fs, \Sigma \vdash_e e_2\colon s_2 \qquad \mathrm{minify}(s_1, s_2) \neq \text{'never}}{fs, \Sigma \vdash_e \mathrm{bop}(e_1, e_2)\colon \mathrm{minify}(s_1, s_2)} \quad \text{(S-BOP)}$$

$$\frac{fs, \Sigma \vdash_e e\colon (\theta, l) \qquad fs, \Sigma \vdash_q q\colon \theta' \qquad \theta \leq \theta'}{fs, \Sigma \vdash_e e\,\texttt{@>}\,q\colon (\theta', l)} \quad \text{(S-SAMPLE)}$$

$$\frac{fs, \Sigma \vdash_e e_1\colon (\theta_1, l_1) \qquad fs, \Sigma \vdash_e e_2\colon (\theta_2, l_2) \qquad \langle f, 2\rangle \leftarrow fs}{fs, \Sigma \vdash_e e_1\,\texttt{@<}\,f\,\texttt{>@}\,e_2\colon (\theta_1 \mid \theta_2, 0)} \quad \text{(S-MERGE)}$$

$$\frac{\langle f, n\rangle \leftarrow fs \qquad \left[fs, \Sigma \vdash_e e_i\colon s_i\right]_{i\in\{1,\ldots,n\}} \qquad \mathrm{minify}(s_1, \ldots, s_n) \neq \text{'never}}{fs, \Sigma \vdash_e f(e_1, \ldots, e_n)\colon \mathrm{minify}(s_1, \ldots, s_n)} \quad \text{(S-APP)}$$

$$\frac{\left[fs, \Sigma \vdash_e e_i\colon s_i\right]_{i\in\{1,\ldots,n\}} \qquad \mathrm{minify}(s_1, \ldots, s_n) \neq \text{'never}}{fs, \Sigma \vdash_e (e_1, \ldots, e_n)\colon \mathrm{minify}(s_1, \ldots, s_n)} \quad \text{(S-TUPLE)}$$

$$\frac{fs, \Sigma \vdash_e e_1\colon s_1 \qquad fs, \Sigma \vdash_e e_2\colon s_2 \qquad fs, \Sigma \vdash_e e_3\colon s_3 \qquad \mathrm{minify}(s_1, s_2, s_3) \neq \text{'never}}{fs, \Sigma \vdash_e \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\colon \mathrm{minify}(s_1, s_2, s_3)} \quad \text{(S-IF)}$$

宣言

$$\frac{\langle v, e\rangle \leftarrow vs \qquad fs, \Sigma \vdash_e v\colon s \qquad fs, \Sigma \vdash_e e\colon s \qquad s \neq \text{'never}}{vs, fs, \Sigma \vdash_s \texttt{node}\ v = e} \quad \text{(S-NODEDEF)}$$

$$\frac{\langle v, n, e\rangle \leftarrow vs \qquad fs, \Sigma \vdash_e v\colon (\theta, n) \qquad fs, \Sigma \vdash_e e\colon (\theta, l)}{vs, fs, \Sigma \vdash_s \texttt{node init}[c_1, \ldots, c_n]\, v = e} \quad \text{(S-NODEDEFINIT)}$$

$$\Sigma' = \{\langle v_1\colon \text{'const}\rangle, \ldots, \langle v_n\colon \text{'const}\rangle\}$$

$$\frac{\langle f, n\rangle \leftarrow fs \qquad fs, \Sigma' \vdash_e e\colon \text{'const} \qquad (e \text{ は時変値演算を含まない})}{vs, fs, \Sigma \vdash_s \texttt{func}\, f(v_1, \ldots, v_n) = e} \quad \text{(S-FUNCDEF)}$$

$$\frac{\left[(v_i, q_i) \in I \qquad fs, \Sigma \vdash_q q_i\colon (\pi_i, \varphi_i) \qquad fs, \Sigma \vdash_e v_i\colon ((\pi_i, \varphi_i), 0)\right]_{i\in\{1,\ldots,n\}}}{vs, fs, \Sigma \vdash_s \texttt{in}\, v_1\colon \tau_1\, q_1, \ldots, v_n\colon \tau_n\, q_n} \quad \text{(S-INDEF)}$$

$$\frac{\left[(v_i, q_i) \in O \qquad fs, \Sigma \vdash_q q\colon (\pi_i, \varphi_i) \qquad fs, \Sigma \vdash_e v\colon ((\pi_i, \varphi_i), l)\right]_{i\in\{1,\ldots,n\}}}{vs, fs, \Sigma \vdash_s \texttt{out}\, v_1\colon \tau_1\, q_1, \ldots, v_n\colon \tau_n\, q_n} \quad \text{(S-OUTDEF)}$$

モジュール

$$\frac{\exists vs\, \exists fs\, \exists \Sigma \Big(vs, fs, \Sigma \vdash_s I \qquad vs, fs, \Sigma \vdash_s O \qquad vs, fs, \Sigma \vdash_s L \qquad \left[vs, fs, \Sigma \vdash_s f\right]_{f\in fs}\Big)}{\vdash_m (I, O, L, fs)} \quad \text{(S-MODULE)}$$

図 11　シーケンスの静的検査のための型判断

マへ提供する関数は以下の通りである．

- タイマーの初期化を行う関数スケルトン
- タイマー割り込みフラグの無効化を行う関数スケルトン
- 割り込みハンドラの実装が記述された関数への参照
- 割り込み待機を行う関数スケルトン
- 各入力時変値を設定する関数スケルトン
- 各出力時変値を設定する関数スケルトン

実際のコード例については，次章のケーススタディで述べる．

5 ケーススタディ

ここでは PbEmfrp を用いた周期的タスクの実装方法と生成されるプログラムを述べていく．ただし，生成されるプログラムの言語は C++を仮定する．

5.1 実装方法

温湿度計を PbEmfrp で実装した場合，ソースコード 3 のようになる．

ソースコード 2 と比べると，時間に関する時変値（`...Timing`）や出力のタイミングであることを知らせる論理値の時変値（`output...`）がなくなっており，代わりの記述としてサンプリング（16 行目，23 行目）が追加されている．もう一つの出力時変値である `display` についても，`tmp` や `hmd` のタイミングを merge によって組み合わせることで，それぞれの更新時に `display` が更新されるようになっている．ソースコード 3 での時間に関する記述は入力時変値やサンプリングに限られており，非常に見通しよく書けている．この意味で，PbEmfrp での記述が（少なくともある種の）周期的タスクの記述に有用であることがわかる．なお，merge のそれぞれのタイミングでは互い違いに@now 参照を行っているが，このような（一方の現在値参照と@now 参照を組み合わせる）処理はタイミングの合成を用いる際に頻出すると思われるため，より容易に記述できる構文を今後用意したい．

5.2 生成されるコードの概観

ソースコード 3 を C++ にコンパイルする場合，処理系は以下の 3 つのプログラムを生成する．

- ThermoHygrometer.cpp
  - 時変値に関する処理が記述された C++プログラム
- ThermoHygrometer.h
  - 上記の 2 ファイル間での関数へのアクセスに用いるヘッダファイル
- ThermoHygrometerMain.cpp
  - main 関数を含む，プログラマがハードウェア依存の処理を記述する C++プログラム

それぞれのファイル名はモジュール名（`ThermoHygrometer`）から付けられる．以下、二つの C++プログラムについて元のプログラムとの対応を簡単に述べる．

5.2.1 時変値に関する処理

ソースコード 4 および 5 は PbEmfrp のプログラムにおける時変値計算やタイミングの処理などを実現したものである．

ソースコード 4 の 4 行目から 12 行目では，各時変値の情報を格納するグローバル変数が定義されている．既に 4.1 節にて述べたとおり，各時変値には発生の値を格納する配列と，最新の値を格納するインデックスの組として定義される．プログラムの経過時刻については，直接タイミングにある数値を使うのではなく，時変値更新の実行間隔 (interval) と FRP ロジックの実行開始時刻 offset およびカウンタによって管理する．ただし，カウンタの値が n である時，プログラムが開始してからの経過時間は $n * \text{interval} + \text{offset}$ で計算されることに注意する．プログラム中においては，このカウンタは 14 行目の `Counter` によって定義され，また interval と offset は 39 行目の `InitTimer` 関数の引数として与えられている．

ソースコード 5 の 63 行目から 90 行目には，割り込み処理が行われた際に実行する内容が `ISR` 関数として定義されている．後述のプログラマが処理を記述する部分からは，この関数を呼ぶことで発生している時変値の更新処理が行われる．各時変値における発生の判定は `ISR` 関数内に記述されており，実際の更新処理は 93 行目から 122 行目に記述されている．

```
module ThermoHygrometer

in  tmp: Float '(5000, 5000), # 温度
    hmd: Float '(3000, 5000)  # 湿度

out sendTmpAve: Float '_,                    # 温度の移動平均
    sendHmdAve: Float '_,                    # 湿度の移動平均
    display   : (Float, Float, Float) '_     # 温度，湿度，不快指数の組

# 温度の移動平均の計算
node init[0.0, 0.0, 0.0] tmpHistory = tmp
node tmpAve = (tmpHistory
            + tmpHistory@last[1]
            + tmpHistory@last[2]
            + tmpHistory@last[3]) / 4.0
node sendTmpAve = tmpAve @> '(30000, 20000)

# 湿度の移動平均の計算
node init[0.0, 0.0] hmdHistory = hmd
node hmdAve = (hmdHistory
            + hmdHistory@last[1]
            + hmdHistory@last[2]) / 3.0
node sendHmdAve = hmdAve @> '(30000, 18000)

# 描画する情報の計算
func calcDisplay((tmp, hmd))  # ディスプレイへの出力を求める関数
  = (tmp, hmd, calcDi(tmp, hmd))
func calcDi(tmp, hmd)         # 不快指数を求める関数
  = 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3
func makePair ((t1, h1), (t2, h2)) = (t1, h2)

node display = calcDisplay((tmp, hmdHistory@now) @< makePair >@ (tmpHistory@now, hmd))
```

ソースコード 3　ThermoHygrometer.pbm

5.2.2　ハードウェア依存の処理

4.3 節で述べたように，タイマーの設定や各入出力時変値に対する処理のようなハードウェア依存の処理は中身がほぼ空の関数スケルトンのみが提供され，そこにプログラマ自身が処理を記述する．ソースコード 3 の場合，この関数群はソースコード 6 のようになる．

特に注意が必要なのは `InitTimer` 関数（23–29 行目）である．この関数はプログラムの初期化の一環として呼び出され（ソースコード 4 の 39 行目），プログラム中の単位時間を作り出す処理になる．この関数でハードウェアタイマーなどを用いて `ISR` 関数が（ハンドラとして、または別途定義したハンドラから）呼び出されるよう設定することで，周期的な処理を実現する．つまり，プログラマはこの周期が適切に処理されるように `InitTimer` 関数を実装することで、全体の処理が進むことが保証される．

6　関連研究

6.1　Fran

Fran [3] は FRP の起源であり，ここでの時変値は連続的に変化する値を表現する Behavior と時刻と値の組を表現する Event という異なる 2 種類のプリミティブから構成されている．このような時変値の表現方法は Emfrp が持つものとは異なるが，本研究で提案する PbEmfrp における離散時変値・連続時変値の意味づけやシーケンスの検査は，Fran の Behavior と Event の意味論に大きく影響を受けている．たとえば，離散時変値と連続時変値間の二項演算は計算結果が離散時変値として得られるという点は，Event の意味論における $snapshot : Event_{\alpha} \to Behavior_{\beta} \to Event_{\alpha \times \beta}$ に対応している．

```cpp
#include "ThermoHygrometer.h"

// 時変値を管理するグローバル変数の宣言
double node_tmp[1]; int node_tmp_index = 0;
double node_hmd[1]; int node_hmd_index = 0;
double node_tmpHistory[4]; int node_tmpHistory_index = 3;
double node_tmpAve[1]; int node_tmpAve_index = 0;
double node_sendTmpAve[1]; int node_sendTmpAve_index = 0;
double node_hmdHistory[3]; int node_hmdHistory_index = 2;
double node_hmdAve[1]; int node_hmdAve_index = 0;
double node_sendHmdAve[1]; int node_sendHmdAve_index = 0;
Tuple3DoubleDoubleDouble node_display[1]; int node_display_index = 0;

int Counter = 0;

// ThermoHygrometerMain.cpp で定義される関数の宣言
extern int Input_tmp();
extern int Input_hmd();
extern void Output_sendTmpAve(double node_sendTmpAve);
extern void Output_sendHmdAve(double sendHmdAve);
extern void Output_display(Tuple3DoubleDoubleDouble display);
extern void InitTimer(int offset_ms, int interval_ms);
extern void ResetInterruptFlag();
[[noreturn]] extern void SleepCPU();
void InitThermoHygrometer();
void ActivateThermoHygrometer();
void MainTask();
Tuple3DoubleDoubleDouble calcDisplay(Tuple2DoubleDouble);
double calcDi(double, double);
Tuple2DoubleDouble makePair(Tuple2DoubleDouble, Tuple2DoubleDouble);

void InitThermoHygrometer() { // ここで全時変値の初期化を行う
  node_tmpHistory[1] = 0.0; node_tmpHistory[2] = 0.0; node_tmpHistory[3] = 0.0;
  node_hmdHistory[1] = 0.0; node_hmdHistory[2] = 0.0;
}

// ThermoHygrometerMain.cpp の main 関数で実行される関数
[[noreturn]] void ActivateThermoHygrometer() {
  InitThermoHygrometer(); InitTimer(1000, 5000); SleepCPU();
}

double calcDi(double tmp, double hmd) {
  return 0.81 * tmp + 0.01 * hmd * (0.99 * tmp - 14.3) + 46.3;
}

Tuple3DoubleDoubleDouble calcDisplay(Tuple2DoubleDouble tmp_hmd) {
  double tmp = tmp_hmd._0; double hmd = tmp_hmd._1;
  return {tmp, hmd, calcDi(tmp, hmd)};
}

Tuple2DoubleDouble makePair(Tuple2DoubleDouble t1_h1, Tuple2DoubleDouble t2_h2) {
  double t1 = t1_h1._0; double h1 = t1_h1._1; double t2 = t2_h2._0; double h2 = t2_h2._1;
  return {t1, h2};
}

double node_hmdHistory_now() { return node_hmdHistory[node_hmdHistory_index]; }

double node_tmpHistory_now() { return node_tmpHistory[node_tmpHistory_index]; }
```

ソースコード 4　**ThermoHygrometer.cpp(前半)**

```
bool cond_1; bool cond_2; bool cond_3; bool cond_4;

// 時変値処理を行う割り込みハンドラ
void ISR() {
  cond_1 = Counter >= 5 && (Counter - 5) % 5 == 0;
  cond_2 = Counter >= 5 && (Counter - 5) % 3 == 0;
  cond_3 = Counter >= 20 && (Counter - 20) % 30 == 0;
  cond_4 = Counter >= 18 && (Counter - 18) % 30 == 0;
  node_tmp_index = (node_tmp_index - cond_1 + 1) % 1;
  node_tmpHistory_index = (node_tmpHistory_index - cond_1 + 4) % 4;
  node_hmd_index = (node_tmpHistory_index - cond_2 + 1) % 1;
  node_tmpAve_index = (node_tmpAve_index - cond_1 + 1) % 1;
  node_sendTmpAve_index = (node_sendTmpAve_index - cond_3 + 1) % 1;
  node_hmdHistory_index = (node_hmdHistory_index - cond_2 + 3) % 3;
  node_hmdAve_index = (node_hmdAve_index - cond_2 + 1) % 1;
  node_sendHmdAve_index = (node_sendHmdAve_index - cond_4 + 1) % 1;
  node_display_index = (node_display_index - (cond_1 || cond_2) + 1) % 1;

  // 入力時変値の処理
  if (cond_1) { node_tmp[node_tmp_index] = Input_tmp(); }
  if (cond_2) { node_tmp[node_tmp_index] = Input_hmd(); }
  // 時変値計算
  MainTask();
  // 出力時変値の処理
  if (cond_3) { Output_sendTmpAve(node_sendTmpAve[node_sendTmpAve_index]); }
  if (cond_4) { Output_sendHmdAve(node_sendHmdAve[node_sendHmdAve_index]); }
  if (cond_1 || cond_2) { Output_display(node_display[node_display_index]); }
  Counter++;
  // 割り込み許可フラグのリセット．
  ResetInterruptFlag();
}

// 入力時変値から出力時変値の計算を行う．
void MainTask() {
  if (cond_1) {
    node_tmpAve[node_tmpAve_index] =
        (node_tmpHistory[node_tmpAve_index] +
         node_tmpHistory[(node_tmpAve_index + 1) % 4] +
         node_tmpHistory[(node_tmpAve_index + 2) % 4] +
         node_tmpHistory[(node_tmpAve_index + 3) % 4]) / 4.0;
    if (cond_3) {
      node_sendTmpAve[node_sendTmpAve_index] = node_tmpAve[node_tmpAve_index];
    }
  }
  if (cond_2) {
    node_hmdAve[node_hmdAve_index] =
        (node_hmdHistory[node_hmdAve_index] +
         node_hmdHistory[(node_hmdAve_index + 1) % 3] +
         node_hmdHistory[(node_hmdAve_index + 2) % 3]) / 3.0;
    if (cond_4) {
      node_sendHmdAve[node_sendHmdAve_index] = node_hmdAve[node_hmdAve_index];
    }
  }
  if (cond_1 || cond_2) {
    Tuple2DoubleDouble _tmp1 = {node_tmp[node_tmp_index], node_hmdHistory_now()};
    Tuple2DoubleDouble _tmp2 = {node_tmpHistory_now(), node_hmd[node_hmd_index]};
    Tuple2DoubleDouble _tmp3;
    if (cond_1 && cond_2) { _tmp3 = makePair(_tmp1, _tmp2); }
    else if (cond_1) { _tmp3 = _tmp1; }
    else if (cond_2) { _tmp3 = _tmp2; }
    node_display[node_display_index] = calcDisplay(_tmp3);
  }
}
```

ソースコード 5　ThermoHygrometer.cpp(後半)

```
#include "ThermoHygrometer.h"
// PbEmfrp 処理系によって生成される，
// 時変値計算を行う ISR 関数はここで
// 参照できるため，割り込みハンドラを
// 設定する際に用いる．

int main() {
  // ここで各種ハードウェアの初期化を行う．
  ActivateThermoHygrometer();
}

// 入力時変値を設定する処理を記述する．
int Input_tmp() {}

int Input_hmd() {}

// 出力時変値を用いた処理を記述する．
void Output_sendTmpAve(double sendTmpAve) {}

void Output_sendHmdAve(double sendHmdAve) {}

void Output_display(Tuple3DoubleDoubleDouble
     display) {}

void InitTimer(int offset_ms, int
    interval_ms) {
  // ここでタイマーの初期化を行う．
  //
  // 引数の offset と interval は PbEmfrp
  // の処理系によって計算されたタイマー割り
  // 込みの開始時刻と周期であり，この値を用
     いてプログラマが
  // タイマーを初期化する．
}

void ResetInterruptFlag() {
  // 設定された割り込みフラグをリセットする
     処理を記述する．
  // この関数は割り込みハンドラの最後に呼ば
     れる．
}

[[noreturn]] void SleepCPU() {
  while (true) {
    // 周期的タスクは割り込みハンドラ内で行
       われるため，
    // ここには消費電力を削減するためにスリ
       ープモードに入る
    // 処理等を記述することができる．
  }
}
```

ソースコード 6　ThermoHygrometerMain.cpp

6.2　Céu

Céu [5] は組込みシステム向け手続き型言語である．この言語では trail と呼ばれる手続きを定義し，これら複数組み合わせてプログラムを記述する．複数の異なる trail は見かけ上スレッドを生成したかのように並行に実行されるが，異なる trail から複雑な記述なく共有状態を安全に扱うことができる点や，処理系はシングルスレッドで動作する C プログラムを生成するといった点が，組込みシステムにおける並行動作する複数のタスクの簡潔な実装を実現している．これに対して PbEmfrp では，複数の周期的タスクはそれらの周期をサンプリングや@now 参照によって明示的に同期させることで値の取得タイミングを指定し，安全な状態共有を実現する．

また，Céu はプログラム内で C 言語の API を利用することができ，副作用を単一のプログラム内で記述できる．しかし PbEmfrp はリアクティブシステムから副作用を分離して純粋なロジックを独立させるようなプログラムの記述を強制している．これにより，ハードウェア依存の副作用を持つ処理に関する移植性の確保や，ロジック単体でのテストを容易にしている．

6.3　Hae

Hae [7] は Haskell をホスト言語とする内部 DSL であり，C++プログラムを生成する．FRP プログラム内に時変値の発生周期を指定することができ，そのコンパイラは各時変値の発生周期と時変値間の依存関係を抽出して解析することで，各イテレーションの周期を最適化している．PbEmfrp では同様の時間解析を行なっているが，タイミングでは最初の発生時刻も考慮されているため，Hae のものより複雑な解析となっている．また，PbEmfrp では初期値を持つ離散時変値から連続的な時変値へと変換する@now 参照が存在しているため，異なる時変値に対して強制的に同期をとる手段が存在するが，Hae にはそのような方法が存在しない点が異なる．

7　結論と今後の課題

本研究では既存の組込みシステム向け FRP 言語である Emfrp に対してシーケンスの概念を導入し，構文・意味・型システム・コード生成の各種に拡張を与える PbEmfrp を提案した．これにより，周期的タスクの簡潔な記述と実行時に計算資源を浪費しないコード生成を可能にすることをケーススタディを通して確認した．

本稿では構文およびシーケンスに関する型判断について形式的に述べ，C++を対象としたコード生成の概要について述べたが，言語処理系の実装には至っておらず，これは今後の課題である．また PbEmfrp は扱う時変値を周期的なものと仮定していたため，入出力の発生時刻が動的に決定されるような非周期的タスクと混合してのプログラムの記述においては，これを考慮したシーケンスの検査および実行モデルを提案できておらず，言語の拡張に余地がある．

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] Courtney, A., Nilsson, H., and Peterson, J.: The Yampa Arcade, *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, New York, NY, USA, Association for Computing Machinery, 2003, pp. 7–18.

[2] Czaplicki, E. and Chong, S.: Asynchronous Functional Reactive Programming for GUIs, *34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2013)*, ACM, 2013, pp. 411–422.

[3] Elliott, C. and Hudak, P.: Functional Reactive Animation, *2nd ACM SIGPLAN International Conference on Functional Programming (ICFP 1997)*, ACM, 1997, pp. 263–273.

[4] Peterson, J., Hudak, P., and Elliott, C.: Lambda in Motion: Controlling Robots with Haskell, *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, Berlin, Heidelberg, Springer-Verlag, 1999, pp. 91—105.

[5] Sant' Anna, F., Ierusalimschy, R., and Rodriguez, N.: Structured Synchronous Reactive Programming with Céu, *Proceedings of the 14th International Conference on Modularity*, MODULARITY 2015, New York, NY, USA, Association for Computing Machinery, 2015, pp. 29–40.

[6] 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, New York, NY, USA, Association for Computing Machinery, 2016, pp. 36–44.

[7] Sheng, W. and Watanabe, T.: Functional Reactive EDSL with Asynchronous Execution for Resource-Constrained Embedded Systems, *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Studies in Computational Intelligence, Vol. 850, Springer, Aug. 2019, pp. 171–190.