組込みシステム向けFRP言語におけるリアルタイムタスクの記述と処理機構

十河 健人　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，時間と共に変化する値である時変値を宣言的に組み合わせることによって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．組込みシステム向け FRP 言語である EvEmfrp は，時変値に対しその更新タイミングを型情報として持たせることで周期的・非周期的なタスク処理を記述することができる．しかし，それぞれの時変値の更新がいつ完了するかといった実時間性に関する保証がなく，リアルタイムシステム向けの言語としては十分ではない．本研究では，EvEmfrp をもとに，リアルタイムシステムを記述するための構文を備えた FRP 言語を提案する．本言語では，明示的にタスクとそのデッドラインを記述することができ，各タスクでどの時変値が更新されるかを時変値が持つタイミングから静的に決定することができる．また，本言語におけるタスク処理に適したスケジューリング機構を提案する．

1　はじめに

リアクティブシステムは外部からの入力に対して応答を行うシステムであり，Web アプリケーションなどの GUI や組込みシステムなどが該当する．リアクティブシステムを手続き型の手法で記述する場合はポーリングやコールバックなどが用いられてきたが，これらの手法を用いるとプログラムが煩雑になりやすいという問題がある [2].

関数リアクティブプログラミング（Functional Reactive Programming，FRP） [10] 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であり，時間と共に変化する値である**時変値**を宣言的に組み合わせることによってシステムを簡潔に記述することができる．

Emfrp [13] は小規模組込みシステム向け FRP 言語である．マイクロコントローラなどリソースが限られた環境で動作させるため，メモリの使用量を静的に見積もることができるなどの特徴を持つ．Emfrp に対し周期的・非周期的なタスクを記述できるように拡張した言語として，EvEmfrp [14] がある．この言語では，時変値ごとにその更新タイミングを型情報として持たせることでそのようなタスクの記述を可能としている．しかし，それぞれの時変値の更新が完了する時刻に対する保証がないため，タスクの実時間性が求められるリアルタイムシステムには適していない．

Priority-based FRP（P-FRP） [12] は，時変値の更新イベントに優先度を導入することで実時間性の考慮を可能とした FRP の派生モデルである．P-FRP の特徴はそのスケジューリング方式である．多くのリアルタイムシステムでは，タスクの実行中に他の優先度の高いタスクが割り込んで（プリエンプトして）実行できるプリエンプティブスケジューリングが用いられ，一般的にはプリエンプトされたタスクは中断した地点から再開することができる．一方 P-FRP では，時変値同士の関係の一貫性を保つため，プリエンプトされたタスクが再開する際に最初からタスクをやり直すという実行モデルが提案された．しかし，このモデルはその性質上タスクの実行が非効率になるという問題がある．

本研究では，EvEmfrp をもとにしたリアルタイムシステム向け FRP 言語である FreacTime を提案す

Real-time Task Description and Processing Mechanism in FRP Language for Embedded Systems

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

る．特徴として明示的に個々のタスクとそのデッドラインを記述することができ，各タスクでどの時変値が更新されるかを時変値が持つタイミングから静的に決定することができる．また，FreacTime におけるタスクモデルに適したスケジューリング機構を提案する．

本論文は以下のように構成される．まず 2 節で本研究の動機について述べる．次に 3 節で FreacTime の言語機能について説明し，4 節でそれらを形式化する．5 節では FreacTime のタスクスケジューリング機構を説明する．そして 6 節で関連研究について述べ，最後に 7 節で本研究のまとめと今後の課題について述べる．

2 研究の動機

リアルタイムシステムとは，それが行う処理（タスク）に対して完了までの期限（デッドライン）が設定されたシステムである．期限までに処理が終わらなかった場合に壊滅的な被害をもたらすようなシステムをハードリアルタイムシステムと呼び，航空機システム，自動車システムや原子力発電所制御システムなどが該当する．これらのシステムでは時間的な保証のほか，システムの機能の正しさや信頼性も重要である [1]．FRP のような宣言的なプログラミング手法は，副作用の分離などを通してプログラムの予測可能性を高めることができるため，リアルタイムシステムのような信頼性が求められるシステムに適していると考える．

EvEmfrp [14] は組込みシステム向けの FRP 言語であり，時変値に自身の更新タイミングを持たせることで周期的・非周期的な処理を記述できる．EvEmfrp の課題として，実時間性の保証があげられる．EvEmfrp のランタイムにおいて，時変値が持つ更新タイミングに対応するのはタイマー割り込みや外部割り込みである．それらの割り込みハンドラでは，どの時変値を更新するべきかを表すイベントをイベントキューに送信する．そしてメインタスクと呼ばれる処理でイベントキューからイベントを取り出し，その時変値を更新する処理を行う．つまり，時変値が持つ更新タイミングはその時変値を更新するべきであることを表すイベントを発行する時刻を表すものであり，実際に更新が開始または完了する時刻に対する保証をするものではない．そのため，実時間性の保証を必要とするリアルタイムシステム向けの言語としては十分ではない．

Priority-based FRP（P-FRP） [12] は FRP に優先度を導入したモデルである．Abort-and-Restart（AR）モデルは P-FRP 向けの実行モデルであり，あるタスクが実行中に他の優先度の高いタスクがプリエンプトして実行する場合，プリエンプトされたタスクが再開する際に最初から実行をやり直すという実行モデルである．このモデルにより時変値間の一貫性を保ちながらタスクのプリエンプションが可能となるが，タスクのやり直しが発生するという性質上実行の効率が悪い．一方，P-FRP であってもタスクで行われる処理によっては一般のプリエンプティブスケジューリングのように中断した地点から再開できる場合もある．例えば，タスク同士が互いに独立している，つまりタスクの実行に関わる時変値が全く異なる場合はプリエンプションが起こってもタスクの再開時に最初からやり直す必要がない．しかし，そのような状況を考慮した実行モデルの議論は行われていない．

そのため本研究では，既存の組込みシステム向け FRP 言語をもとにしてリアルタイムシステム向け FRP 言語を設計し，タスク同士の独立性を形式的に表現し，それを用いたスケジューリング機構を提案する．

3 FreacTime

FreacTime は本研究で提案するリアルタイムシステム向け FRP 言語である．この言語は Emfrp をベースにし，EvEmfrp の要素を取り入れながら，リアルタイムシステム向けに明示的なタスクの記述とそのデッドラインの設定ができるように設計された言語である．本節では FreacTime の言語機能について説明する．

3.1 サンプルコード

FreacTime のサンプルコードを図 1 に示す．このプログラムは簡易的な自動ドア制御システムを記述したものである．通行人を検知するセンサーが前後

に 2 つあり（sensor1，sensor2），これらの状態とドアの開閉状態（pos）に応じてドアをどう動かすか（direction）を決める．また，障害物を検知するセンサーもあり，ドアが閉じている状態で障害物を検知した場合はドアを開くように制御する．この制御は人などが挟まれることを防ぐためのもので，実時間性が求められる．

以降，このサンプルコードを例に FreacTime の言語機能について説明する．

3.2 時変値

Emfrp と同様に FreacTime における時変値はノードと呼ばれる．ノードには入力ノードと状態ノードの 2 種類がある．

入力ノードはキーワード inに続いて宣言されるノードであり，外部からの入力を値とする時変値である．例題では，sensor1，sensor2，control，pos，obstacleが入力ノードである．その中で例えば 2，3 行目の sensor1と sensor2は，それぞれドアの前後に設置されたセンサーからの入力を値とする．入力ノードはそれぞれ入力タイミングという情報を持つ．入力タイミングは外部からの入力を受け取るタイミングを抽象化したものであり，具体的なタイミングは後述のタスクによって指定される．入力ノードのタイミングは，そのノードの宣言でデータ型以外に何も指定されていない場合はノード名に「'」をつけたもので表現される．例えば，sensor1は 'sensor1という入力タイミングを持つ．一方 5 行目の posのようにデータ型の後にタイミングが指定されている場合はそのタイミングを持つ．また，4 行目，6 行目のようにデータ型として Unitが指定されているものがある．これは外部からの入力を受け取らないが，ある種のイベントとしてタイミングだけを使うために定義される．

状態ノードはキーワード nodeに続いて定義されるノードであり，システムの中間状態を表す時変値である．例題では，maxPos，minPos，activate，closed，open，closing，startClosing，directionが状態ノードである．状態ノードもタイミングをもち，これはノード同士の依存関係によって導出される．例えば 13 行目の closedは posに依存しており，posのタイミングが 'controlであるため closedもタイミング 'controlをもつ．状態ノードには初期値を設定することができ，初期値を持つノードは@lastを用いて過去に更新された時の値を参照することができる．例えば 17 行目のノード startClosingはノード openの現在の値と過去 2 回の値を用いて定義されており，ドアが開いている状態を 3 回確認した時にドアを閉じるように制御するということを表現している．

3.3 タイミング

ノードはそれぞれタイミングを持つと説明してきた．このタイミングは，具体的には定数であることを表す *const* か，入力タイミングか入力タイミングの列で表される．例えば 8 行目の maxPosのタイミングは *const* であり，後に説明するが 11 行目の activateは入力タイミングの列 'sensor1 | 'sensor2をタイミングとして持つ．タイミングはどの入力時変値と同時に更新されるかを表す．例えば activateは sensor1または sensor2が更新された時に同時に更新される．

3.4 式

ノードの定義には二項演算や if 式などのを用いることができる．このような式を基本式と呼ぶ．二項演算などで異なるタイミングを持つノードを組み合わせる場合，その式のタイミングはそれぞれのタイミングを合成したようなタイミングになる．例えば activateの定義式となっている sensor1 || sensor2は，sensor1と sensor2のタイミングを合成した 'sensor1 | 'sensor2を持つ（そのため activateのタイミングも 'sensor1 | 'sensor2となる）．

when q { e }の形で記述される式を when 式と呼ぶ．q はタイミングアノテーションと呼ばれるもので，ここでは when 式の評価が行われるタイミングを指定する．例えば 24 行目の when 式は，25 行目から 29 行目までの式をタイミング 'controlに評価する式である．

@@はタイミング和と呼ぶ演算子で，@@の前の式が評価されるタイミングではその式が，そうでない場合は後の式が式全体の結果になる．例えば，directionの値はタイミング 'obstacleでは 22，23 行目を評価し

```
1  module AutomaticDoorControl
2  in sensor1 : Bool,
3     sensor2 : Bool,
4     control : Unit,
5     pos     : Int 'control,
6     obstacle : Unit
7
8  node maxPos = 100
9  node minPos = 0
10
11 node activate = sensor1 || sensor2
12
13 node closed = pos == minPos
14 node init[false, false] open = pos == maxPos
15 node closing = direction@last[1] < 0
16
17 node startClosing = open@last[2] &&
18   open@last[1] && open
19
20 # 0: stop, 1: open, -1: close
21 node init[0] direction = when 'obstacle {
22   if closing then 1
23   else direction@last[1]
24 } @@ when 'control {
25   if activate then if open then 0 else 1
26   else if closed then 0
27   else if startClosing then -1
28   else if open then 0
29   else direction@last[1]
30 }
31
32 init 'sensor1[sensor1=false],
33   'sensor2[sensor2=false],
34   'control[pos=0]
35
36 task checkActSensor1 = periodic(0s, 500ms) {
37   in 'sensor1
38   deadline 500ms
39 }
40 task checkActSensor2 = periodic(0s, 500ms) {
41   in 'sensor2
42   deadline 500ms
43 }
44 task controlDoor = periodic(0ms, 200ms) {
45   in 'control
46   out direction
47   deadline 200ms
48 }
49 task detectObstacle = interrupt(500ms) {
50   in 'obstacle
51   out direction
52   deadline 100ms
53 }
```

図 1　AutomaticDoorControl.frtm

た値に，タイミング'controlでは 25 行目から 29 行目までを評価した値になる．

3.5　初期化子

状態ノード activateはタイミング'sensor1 | 'sensor2を持つと説明した．ここで，最初に sensor1が更新され，その時にまだ一度も sensor2が更新されていないような場合を考える．この時，sensor2の値はまだ定まっていないため，activateの値も定まらなくなってしまう．このように，ある時変値がそれが更新されないタイミングで参照される場合に値がまだ定まっていないような状態を避けるために，initキーワードを用いて初期化子を記述することでノードの初期値を設定することができる．コード例では 32 行目から 34 行目までで入力ノードを初期化している．この時，初期化されたノードに依存するノードも定義式に従って初期化される．

3.6　タスク

ノードはそれぞれタイミングを持つが，これはどの入力ノードが更新された時に自身も更新されるのかということを表す抽象的な情報である．どのノードが具体的にいつ更新されるのかは，タスクとして記述される．

タスクが実行される具体的な時刻はイベントによって指定される．イベントは3種類あり，まず`periodic`(`t`, `t'`)は開始時刻 t，周期 t' の周期的イベントを表す．たとえば，36 行目のタスク `checkActSensor1`は`periodic(0s, 500ms)`となっており，通行人を検知するセンサーを 500ms ごとにチェックするタスクと定義されている．`interrupt`(t)は外部割り込みによる散発的イベントを表し，t はイベントの発生の最小インターバルを表す．４９行目のタスク `detectObstacle`のイベントは外部割り込みのイベントとなっており，障害物を検知するセンサーが障害物を検知した時に発生する．最後に `event`(τ`->`t)はタスク τ から時刻 t 後に発生する内部イベントを表す．

タスクにおいてどのノードが更新されるかは，`in`に続いて記述されるタイミングアノテーションで指定される．例えば，タスク `controlDoor`ではタイミング`'control`が指定されている．そのためまずタイミング`'control`を持つ入力ノード `pos`が更新され，その後タイミング`'control`を持つ状態ノードである `closed`や`direction`などが更新される．`out`に続いて記述されているノードは，タスクの最後に外部へ出力するノードを表す．`deadline`キーワードはタスクのデッドラインを指定する．

例題における各タスクの動作は 2 のようになる．

4 形式化

3 節では本研究で提案する言語 FreacTime の機能を説明した．本節では FreacTime の形式化を行う．この形式化は，5 節でスケジューリングについての議論を行うために必要となる．

まず最初に FreacTime の構文を定義する．そして次にタイミングの定義とそれぞれのノードが持つタイミングの導出について述べる．その後，依存・参照関係という概念を導入する．これはノード間の関係を表すもので，タスクの形式化のために必要となる．最後に個々のタスクがどのように実行されるかを形式的に定義する．

4.1 構文

FreacTime の構文は図 3 のように定義される．

注意点としては，`@@`はノード定義のトップレベルでのみ記述でき，`when`式も二項演算などの部分式として記述することはできない．これは，タイミングに関する意味論を明確にするためである．例えばそれぞれ異なるタイミングをもつノードを用いて`(a @@ b) + c`という式が書けた場合，`c`が更新されるタイミングでは `a`と `b`のどちらを参照すべきかが曖昧になってしまう．

4.2 タイミングの形式化

タイミングは図 4 のように定義される．ノードのタイミング Θ は，定数であることを示す $const$ か，入力ノードのタイミング θ の列 $\theta_1, \ldots, \theta_m$ で表される．このとき，それぞれ同じ入力タイミングで構成されるが並び順が異なるような 2 つのタイミングは同じタイミングとみなす．

$\in_\Theta$ は入力タイミング θ がタイミング Θ に含まれることを表す．$\sqsubseteq_\Theta$ はタイミングに関する半順序関係であり，タイミング Θ_1 がタイミング Θ_2 に含まれることを表す．merge は複数のタイミングを合成する関数である．例えば，$\mathrm{merge}\{\theta_1 \mid \theta_2, \theta_3\} = \theta_1 \mid \theta_2 \mid \theta_3$ である．この merge 関数は，タイミングの半順序関係 $\sqsubseteq_\Theta$ における上限として定義することができる．

それぞれのノードがどのタイミングを持つかという情報を持つものをタイミング環境と呼び，Γ で表す．タイミング環境 Γ の導出は以下の手順で行われる．

1. すべての状態ノード n^s に対して $\Gamma(n^s) = const$ となるように Γ を初期化する．
2. 入力ノードのタイミングを求める．入力ノード n^i が $n^i : \rho$ と定義されている場合は，n^i のタイミングは $'n^i$，つまり $\Gamma(n^i) = 'n^i$ となるよう Γ を更新する．n^i が $n^i : \rho\ 'n^i_q$ と定義されている場合，n^i_q が $n^i_q : \rho \in \mathcal{I}$ と定義されているなら $\Gamma(n^i) = 'n^i_q$ とし，そうでない場合はタイミングエラーとする．
3. ノード同士の依存関係から有向グラフを構築する．この時，ノード n^s の更新式に現れる式の種類によって以下のように依存関係を定義する．
   - `when` q `{` e^b `}`の場合，n^s は q で指定された入力ノードに依存するとする．

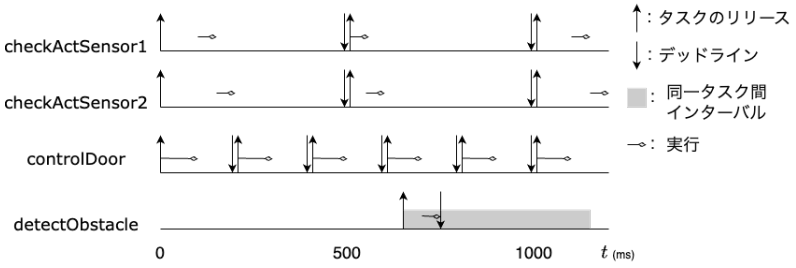

図 2　図 1 のタスクの動作

$$
\begin{array}{llllllllllll}
\mathit{StateNode} & \ni & n^s & \mathit{InputNode} & \ni & n^i & \mathit{Variable} & \ni & x & \mathit{Literal} & \ni & l \\
\mathit{Function} & \ni & f & \mathit{UnaryOp} & \ni & uop & \mathit{BinaryOp} & \ni & bop & \mathit{ConstrLabel} & \ni & C \\
\mathit{DataType} & \ni & \rho & \mathit{Time} & \ni & t & \mathit{Task} & \ni & \tau & \mathit{Module} & \ni & M
\end{array}
$$

$$
\begin{array}{lllll}
\mathit{Node} & \ni & n & ::= & n^s \mid n^i \\
\mathit{TimingAnnotation} & \ni & q & ::= & \text{'}n_1^i \mid \ldots \mid \text{'}n_m^i \\
\mathit{Expression} & \ni & e & ::= & e^w \mid e \texttt{ @@ } e^w \\
\mathit{WhenExpression} & \ni & e^w & ::= & e^b \mid \texttt{when } q \texttt{ \{ } e^b \texttt{ \}} \\
\mathit{BasicExpresison} & \ni & e^b & ::= & l \mid x \mid n \mid n^s\texttt{@last[}l\texttt{]} \mid uop\ e^b \mid e_1^b\ bop\ e_2^b \\
 & & & \mid & f\texttt{(}e_1^b, \ldots, e_m^b\texttt{)} \mid C\texttt{(}e_1, \ldots, e_m\texttt{)} \mid \texttt{if } e_1^b \texttt{ then } e_2^b \texttt{ else } e_3^b \\
 & & & \mid & \texttt{case } e^b \texttt{ of } p_1 \texttt{ -> } e_1^b, \ldots, p_m \texttt{ -> } e_m^b \\
\mathit{Pattern} & \ni & p & ::= & C\texttt{(}x_1, \ldots, x_m\texttt{)} \\
\mathit{Event} & \ni & \epsilon & ::= & \texttt{periodic(}t_1, t_2\texttt{)} \mid \texttt{interrupt(}t\texttt{)} \mid \texttt{event(}\tau\texttt{->}t\texttt{)}
\end{array}
$$

$$
\begin{array}{llll}
\mathcal{M} & ::= & \texttt{module } M \texttt{ in } \mathcal{I}\ \ \mathcal{F}\ \ \mathcal{N}\ \ \texttt{init } \mathit{Init}\ \ \mathcal{T} & \text{(module definition)} \\
\mathcal{I} & ::= & \varnothing \mid \mathcal{I}, n^i \texttt{ : } \rho \mid \mathcal{I}, n^i \texttt{ : } \rho\ \text{'}n_q^i & \text{(input node declarations)} \\
\mathcal{F} & ::= & \varnothing \mid \mathcal{F}, \texttt{fun } f\texttt{(}x_1, \ldots, x_m\texttt{) = } e^b & \text{(function definitions)} \\
\mathcal{N} & ::= & \varnothing \mid \mathcal{N}, \texttt{node } n^s \texttt{ = } e \mid \mathcal{N}, \texttt{node init[}e_1, \ldots, e_m\texttt{]} \ n^s \texttt{ = } e & \text{(node definitions)} \\
\mathit{Init} & ::= & \varnothing \mid \mathit{Init}, \text{'}n^i \mid \mathit{Init}, \text{'}n^i \texttt{ [}n_1^i\texttt{=}l_1, \ldots, n_m^i\texttt{=}l_m\texttt{]} & \text{(initializer definitions)} \\
\mathcal{T} & ::= & \varnothing \mid \mathcal{T}, \texttt{task } \tau \texttt{ = } \epsilon \texttt{ \{ in } q \texttt{ out } n_1^s, \ldots, n_m^s \texttt{ deadline } t \texttt{ \}} & \text{(task definitions)}
\end{array}
$$

図 3　FreacTime の構文

$$
\begin{array}{llllll}
\mathit{InputTiming} & \ni & \theta & ::= & \text{'}n^i & (n^i \in \{n^i \mid n^i\!:\!\rho \in \mathcal{I}\}) \\
\mathit{Timing} & \ni & \Theta & ::= & \theta_1 \mid \cdots \mid \theta_m & (m \geq 1) \\
 & & & \mid & \mathit{const} &
\end{array}
$$

$$
\begin{array}{lll}
\in_\Theta & = & \{(\theta, \Theta) \mid \exists \theta_1, \ldots, \theta_m.(\Theta = \theta_1 \mid \cdots \mid \theta_m \wedge \exists i.\theta_i = \theta)\} \\
\sqsubseteq_\Theta & = & \{(\Theta_1, \Theta_2) \mid \ \forall \theta.(\theta \in_\Theta \Theta_1 \Rightarrow \theta \in_\Theta \Theta_2)\} \\
\mathrm{merge}\{\Theta_1, \ldots, \Theta_m\} & = & \sup\{\Theta_1, \ldots, \Theta_m\}
\end{array}
$$

$$\Gamma \quad : \quad \mathit{Node} \to \mathit{Timing} \quad \text{(タイミング環境)}$$

図 4　タイミングの定義

- それ以外の場合，n^s は@lastによる参照を含みその式に現れるノードに依存するとする．

4. ある入力ノード n^i について，n^i から依存グラフをたどり到着可能なノード n^s に対して $\Gamma(n^s) = \mathrm{merge}\{\Gamma(n^s), \Gamma(n^i)\}$ となるよう Γ を更新する．この操作を n^i から到着可能なノード

全てに対して行う．

5. 4 をすべての入力ノードに対して逐次的に行う．

以降 Γ はこの手順によって導出されたタイミング環境を指す．

4.3 依存・参照関係

それぞれ異なるタイミングをもつノード a，b を用いて，ノード c が

$$c = a + b$$

と定義されているとする．a が更新された時，c は a の変化に応じて更新される．この時，c は a だけでなく b も定義に用いられているため，c は変化した a の値と b の現在の値を用いて更新される．このような場合，c は a に依存して変化し，その時 b を参照していると考えることができる．このように，ノード間の関係には依存関係の他に参照関係も存在する．

ノード間の関係は，

- そのノードの変化に自身が依存しているという関係を表す**依存**，
- そのノードの直前値を参照しているという関係を表す**直前値参照**，
- そのノードの値を参照しているという関係を表す**参照**

の 3 つの関係で定義される．ただし直前値の参照は `@last`による参照のことを指す．これらの関係はタイミングによって変化する．先ほどの例では，a が更新されるタイミングでは c は a に依存し b を参照しているが，b が更新されるタイミングでは c は a を参照し b に依存している．このようなノード間の関係を**依存・参照関係**と呼び，図 5 のように定義する．

r はあるノードに対しての特定のタイミング θ における依存・参照関係を表し，タイミング θ，その時の依存ノード，直前値参照ノード，参照ノードの 4 つ組で表される．依存・参照ノード群 δ は r の集合で表され，あるノードに対してそれぞれのタイミングでどのようなノードと依存・参照関係を持つかを表す．依存・参照環境 Δ は全てのノードに対する依存・参照関係を表し，ノードから依存・参照ノード群への写像として表現される．

依存・参照関係の導出のための関数を図 6 に示す．

$$
\begin{array}{rcll}
r & ::= & (\theta, N^d, N^l, N^r) & \\
N^d & : & \text{依存ノード} & \\
N^l & : & \text{直前値参照ノード} & \\
N^r & : & \text{参照ノード} & \\
\delta & ::= & \varnothing \mid \delta, r & \text{(依存・参照ノード群)} \\
\Delta & ::= & \varnothing \mid \Delta, n^s \mapsto \delta & \text{(依存・参照環境)}
\end{array}
$$

図 5　依存・参照関係

$\mathrm{nodes}(e^b)$ は基本式 e^b の中に現れるノードの集合を得る関数であり，$\mathrm{nodes}_{\mathrm{lst}}(e^b)$ は e^b の中で`@last`参照されるノードの集合を得る関数である．これらは基本式の構造から帰納的に定義することができる．$\mathrm{timing}(e^b, \Gamma)$ は基本式 e^b のタイミングを求める関数であり，$\mathrm{timing}_{\mathrm{q}}(q, \mathcal{I})$ はタイミングアノテーション q が表すタイミングを求める関数である．そして $\mathrm{Rel}[\![e]\!](\mathcal{I}, \Gamma)$ は式 e に対する依存・参照ノード群 δ を求める関数である．

これらの関数を用いて，ノード定義 $\mathcal{N}$ に対する依存・参照環境 Δ は，$\Delta = \{n^s \mapsto \mathrm{Rel}[\![e]\!](\mathcal{I}, \Gamma) \mid n^s = e \in \mathcal{N} \vee$ `node init[`$e_1, \ldots, e_m$`] =` $e \in \mathcal{N}\}$ と導出される．以降 Δ はこのようにして導出された依存・参照環境を指すものとする．

依存・参照関係を用いてラベル付き有向グラフを構築すれば，システムのタイミング毎のノード間の依存関係を視覚的に表現することができる．また，依存・参照関係は次の 4.4 節でタスクの形式的な定義にも用いる．

4.4 タスクの形式化

タスクは外部からの入力を受け取り，それに依存するノードを更新し，特定のノードの値を外部へ出力する処理とみなすことができる．これを形式的に表すと図 7 のようになる．以降，これについて詳しく説明する．

まず，タスクにおいて行われる処理に関係するノードについて整理する．どの入力ノードの値を外部から受け取るかは，タイミングアノテーション q によって決まる．q で指定されたタイミングを Θ_τ とすると，$\Theta_\tau = \mathrm{timing}_{\mathrm{q}}(q, \mathcal{I})$ である．そして，更新するべき入力ノードの集合を I_τ とすると，

$$
\begin{array}{lcl}
\mathrm{nodes}(e^b) & : & e^b \text{ の中に現れるノードの集合を得る関数} \\
\mathrm{nodes}_{\mathrm{lst}}(e^b) & : & e^b \text{ の中で } \texttt{@last} \text{参照されるノードの集合を得る関数} \\
\mathrm{timing}(e^b, \Gamma) & = & \mathrm{merge}\{\Gamma(n) \mid n \in (\mathrm{nodes}(e^b) \cup \mathrm{nodes}_{\mathrm{lst}}(e^b))\} \\
\mathrm{timing}_{\mathrm{q}}(\text{'}n^i, \mathcal{I}) & = & \begin{cases} \text{'}n^i & \text{if } \exists \rho. n^i : \rho \in \mathcal{I} \\ \text{'}n^i_q & \text{if } \exists \rho, \rho'. (n^i : \rho\ \text{'}n^i_q \in \mathcal{I} \wedge n^i_q : \rho' \in \mathcal{I}) \end{cases} \\
\mathrm{timing}_{\mathrm{q}}(\text{'}n^i_1 \mid \ldots \mid \text{'}n^i_m, \mathcal{I}) & = & \mathrm{merge}\{\mathrm{timing}_{\mathrm{q}}(\text{'}n^i_1, \mathcal{I}), \mathrm{timing}_{\mathrm{q}}(\text{'}n^i_2 \mid \ldots \mid \text{'}n^i_m, \mathcal{I})\}
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{Rel}[\![e\ \texttt{@@}\ e^w]\!](\mathcal{I}, \Gamma) & = & \delta \cup \{r \mid r \in \mathrm{Rel}[\![e^w]\!](\mathcal{I}, \Gamma) \wedge \forall r' \in \delta.\pi_1(r) \neq \pi_1(r)\} \\
\quad \text{where } \delta = \mathrm{Rel}[\![e]\!](\mathcal{I}, \Gamma) & & \\
\mathrm{Rel}[\![\texttt{when}\ q\ \texttt{\{}\ e\ \texttt{\}}]\!](\mathcal{I}, \Gamma) & = & \{(\theta, N^d(\theta), N^l(\theta), N^r(\theta)) \mid \theta \in_\Theta \Theta\} \\
\quad \text{where} \quad \Theta & = & \mathrm{timing}_{\mathrm{q}}(q, \mathcal{I}) \\
\quad N^d(\theta) & = & \{n \mid n \in \mathrm{nodes}(e) \wedge \theta \in_\Theta \Gamma(n)\} \\
\quad N^l(\theta) & = & \{n^s \mid n^s \in \mathrm{nodes}_{\mathrm{lst}}(e) \wedge \theta \in_\Theta \Gamma(n)\} \\
\quad N^r(\theta) & = & \{n \mid n \in (\mathrm{nodes}(e) \cup \mathrm{nodes}_{\mathrm{lst}}(e)) \wedge (\theta \in_\Theta \Gamma(n) \vee \Gamma(n) = \mathit{const})\} \\
\mathrm{Rel}[\![e^b]\!](\mathcal{I}, \Gamma) & = & \{(\theta, N^d(\theta), N^l(\theta), N^r(\theta)) \mid \theta \in_\theta \Theta\} \\
\quad \text{where} \quad \Theta & = & \mathrm{timing}(e^b, \Gamma) \\
\quad N^d(\theta) & = & \{n \mid n \in \mathrm{nodes}(e^b) \wedge \theta \in_\Theta \Gamma(n)\} \\
\quad N^l(\theta) & = & \{n^s \mid n^s \in \mathrm{nodes}_{\mathrm{lst}}(e^b) \wedge \theta \in_\Theta \Gamma(n)\} \\
\quad N^r(\theta) & = & \{n \mid n \in (\mathrm{nodes}(e^b) \cup \mathrm{nodes}_{\mathrm{lst}}(e^b)) \wedge (\theta \in_\Theta \Gamma(n) \vee \Gamma(n) = \mathit{const})\}
\end{array}
$$

図 6　依存・参照関係の導出

$I_\tau = \{n^i \mid \Gamma(n^i) \sqsubseteq_\Theta \Theta_\tau\}$ となる．

次に，更新するべき状態ノードを考える．これらは更新された入力ノードに依存するノードであるが，タスクのタイミング Θ_τ によって求められる．更新するべき状態ノードの集合を U_τ とすると，$U_\tau = \{n^s \mid \exists \theta \in_\Theta \Theta_\tau. \theta \in_\Theta \Gamma(n^s)\}$ となる．

状態ノードの更新には，入力ノード以外にも状態ノードの更新式で参照されているノードも必要である．これらは，直前値を参照されるノードと，q によって指定されたタイミングを持たない参照ノードの集合であり，依存・参照関係 Δ によって求めることができる．まず直前値を参照するノードは，更新するべきノードの集合 U_τ のうちのいずれかのノードの更新式で，タイミング Θ_τ で直前値参照されるノードの集合なので，$\{n^s \mid n^{s'} \in U_\tau. \exists \delta \in \Delta(n^{s'}). \exists r \in \delta.\pi_1(r) \in_\Theta \Theta_\tau \wedge n^s = \pi_2(r)\}$ となる．次に，参照ノードの集合を求める．これは，U_τ のうちのいずれかのノードの更新式で，タイミング Θ_τ で参照されるノードのうち，タスク τ で更新されないノードの集合である．これを形式的に表すと，$\{n \mid \exists n^s \in U_\tau. \exists \delta \in \Delta(n^s). \exists r \in \delta.\pi_1(r) \in_\Theta \Theta_\tau \wedge n \in \pi_3(r)\} \setminus (I_\tau \cup U_\tau)$ となる．よって参照ノードの集合 R_τ はこれらの和集合となる．

外部に出力するノードの集合 O_τ は，タスクに記述されたノードの集合なので $O_\tau = \{n^s_1, \ldots, n^s_m\}$ となる．

次に，タスクで行われる処理を形式的に表現する．タスクの実行は以下の 5 つのフェーズからなる．

1. 入力フェーズ
2. コピーフェーズ
3. 更新フェーズ
4. 書き戻しフェーズ
5. 出力フェーズ

入力フェーズでは，外部から入力を受け取り新たな入力ノードの状態を得る．ここでは状態は，ノードからそれが取りうる値への写像として表現する．入力ノードの状態を $\sigma^i_\tau : I_\tau \to V$（$V$ はノードが取りうる値の集合）とし，外部から入力を受け取る関数を

$$
\begin{aligned}
&\texttt{task}\ \tau = \epsilon\ \{\ \texttt{in}\ q\ \texttt{out}\ n^s_1, \ldots, n^s_m\ \texttt{deadline}\ t\ \}\\
&\Theta_\tau \quad = \quad \mathrm{timing}_{\mathrm{q}}(q, \mathcal{I})\\
&I_\tau \quad = \quad \{n^i \mid \Gamma(n^i) \sqsubseteq_\Theta \Theta_\tau\}\\
&U_\tau \quad = \quad \{n^s \mid \exists\theta \in_\Theta \Theta_\tau.\theta \in_\Theta \Gamma(n^s)\}\\
&R_\tau \quad = \quad \{n^s \mid n^{s\prime} \in U_\tau.\exists\delta \in \Delta(n^{s\prime}).\exists r \in \delta.\\
&\qquad\quad \pi_1(r) \in_\Theta \Theta_\tau \wedge n^s = \pi_2(r)\}\cup\\
&\qquad\quad (\{n \mid \exists n^s \in U_\tau.\exists\delta \in \Delta(n^s).\exists r \in \delta.\\
&\qquad\quad \pi_1(r) \in_\Theta \Theta_\tau \wedge n \in \pi_3(r)\} \setminus (I_\tau \cup U_\tau))\\
&O_\tau \quad = \quad \{n^s_1, \ldots, n^s_m\}
\end{aligned}
$$

$$
\begin{array}{llllll}
\sigma^i_\tau & : & I_\tau \to V & \sigma^r_\tau & : & R_\tau \to V\\
\sigma^u_\tau & : & U_\tau \to V & \sigma^o_\tau & : & O_\tau \to V\\
\sigma & : & \mathit{Node} \to V & & & \\
\mathrm{input}_\tau & : & () \mapsto \sigma^i_\tau & & & \\
\mathrm{update}_\tau & : & \sigma^i_\tau \times \sigma^r_\tau \mapsto \sigma^u_\tau & & & \\
\mathrm{output}_\tau & : & \sigma^o_\tau \mapsto () & & &
\end{array}
$$

$$
\begin{aligned}
&\mathit{task}_\tau = \{\\
&\quad \sigma^i_\tau \quad \leftarrow \mathrm{input}_\tau();\\
&\quad \sigma^r_\tau \quad \leftarrow \{n \mapsto \sigma(n) \mid n \in R_\tau\};\\
&\quad \sigma^u_\tau \quad \leftarrow \mathrm{update}_\tau(\sigma^i_\tau, \sigma^r_\tau);\\
&\quad \sigma \quad \leftarrow \sigma^i_\tau \cup \sigma^u_\tau \cup \{n \mapsto \sigma(n) \mid n \notin (I_\tau \cup U_\tau)\};\\
&\quad \sigma^o_\tau \quad \leftarrow \{n \mapsto \sigma(n) \mid n \in O_\tau\};\\
&\quad \mathrm{output}_\tau(\sigma^o_\tau);\\
&\}
\end{aligned}
$$

図 7　タスクの実行

input_τ とすると，入力フェーズは $\sigma^i_\tau \leftarrow \mathrm{input}_\tau()$ と表せる．

コピーフェーズでは，システム全体の状態から参照ノードの状態をローカルにコピーする．システム全体の状態を $\sigma : \mathit{Node} \to V$，参照ノードの状態を $\sigma^r_\tau : R_\tau \to V$ とすると，コピーフェーズは $\sigma^r_\tau \leftarrow \{n \mapsto \sigma(n) \mid n \in R_\tau\}$ と表せる．

更新フェーズでは．新たに得た入力ノードの状態と参照ノードの状態から更新するノードの状態を求める．更新するノードの状態を $\sigma^u_\tau : U_\tau \to V$ とし，状態ノードの値を更新する関数を update_τ とすると，更新フェーズは $\sigma^u_\tau \leftarrow \mathrm{update}_\tau(\sigma^i_\tau, \sigma^r_\tau)$ と表せる．

書き戻しフェーズでは，新しい入力ノードの状態と更新した状態ノードの値をシステム全体の状態に書き戻す．これは，$\sigma \leftarrow \sigma^i \cup \sigma^u_\tau \cup \{n \mapsto \sigma(n) \mid n \notin (I_\tau \cup U_\tau)\}$ と表せる．

最後に，出力フェーズでは，出力ノードの状態を外部に出力する．出力ノードの状態を $\sigma^o_\tau : O_\tau \to V$ とし，出力関数を output_τ とすると，出力フェーズは $\sigma^o_\tau \leftarrow \{n \mapsto \sigma(n) \mid n \in O_\tau\}; \mathrm{output}_\tau(\sigma^o_\tau)$ と表せる．

5　スケジューリング

3 節では，FreacTime のプログラムがどのように記述され，個々のタスクがどのように動作するかを述べてきた．本節では，リアルタイムシステムで求められる時間的な制約を満たすためにどのようにタスクを協調させるかについて述べる．

5.1　タスクに関する表記法

本研究は，単一のプロセッサ上で実行される固定優先度のハードリアルタイムシステムを対象とする．システムは FreacTime のプログラムで定義された m 個のタスクからなり，$\tau_1, \ldots, \tau_m$ と表記する．この時，各タスクは優先度の高い順に並べられているとする．

タスクは周期的もしくは散発的（非周期的）に何度も実行されるが，それぞれの実行をジョブと呼び，タスク τ_i の k 番目のジョブを J^k_i と表記する．タスクはいくつかのパラメータによって特徴付けられる．まずはタスク τ_i の（最悪）実行時間で，これは $\mathbf{C}_i$ と表記する．タスクが完了するべき時刻はデッドラインと呼ばれ，$\mathbf{D}_i$ と表記する．そして周期（散発的タスクの場合はジョブ間の最小インターバル）は $\mathbf{T}_i$ と表記する．また，タスクには優先度が割り当てられており，これは $\mathbf{P}_i$ と表記する．タスクのスケジューリングの際には，これらのパラメータを考慮して行われる．

実行時間 $\mathbf{C}$ や周期 $\mathbf{T}$ といったパラメータは非負整数値を取るとする．つまり，一つのリアルタイムクロックがあり，そのクロックサイクルを時間の最小単位としてシステムは動作することとする．

5.2　プリエンプティビリティ

2 節で述べたように，FRP では時変値間の関係によってシステムを記述するため，マルチタスクを行う場合には時変値間の一貫性を保つ必要がある．そのため P-FRP では，Abort-and-Restart（AR）モデル

やノンプリエンプティブスケジューリング（Nonpreemptive Scheduling, NP）を用いてきた．しかしこれらのモデルは一般のプリエンプティブスケジューリング（Preemptive Scheduling, PS）に比べ，高優先度のタスクがデッドラインミスを犯しやすいという問題がある．

一方，個々のタスクに注目すると，そのタスクで参照・更新される時変値によってはプリエンプションされても AR モデルのようにタスクの再開時に最初からやり直す必要がない場合がある．例として，ソースコード 1 のシステムを考える．タスク `checkActSensor1`は，500ms ごとに通行人が検知されたかを確認し，ノード `activate`を更新する．このタスクが実行中にタスク `detectObstacle`がリリースされたとする．このタスクは障害物センサーが障害物を検知した場合にリリースされ，ドアが閉じている場合にはノード `direction`を更新しドアを開くように制御する．ノード `activate`はノード `direction`の値を参照しておらず，タスク `detectObstacle`におけるノード `direction`の更新はノード `activate`の値を用いない．そのため，タスク `checkActSensor1`とタスク `detectObstacle`は互いに独立しており，タスク `checkActSensor1`の実行にタスク `detectObstacle`がプリエンプとしても，`checkActSensor1`は中断した地点から再開しても良い．

このように，あるタスク τ_l が別の高優先度のタスク τ_h にプリエンプトされても良い（再開時に最初からやり直す必要がない）場合，τ_l は τ_h に対しプリエンプティブルであると呼ぶことにする．具体的に定義すると，

定義 1. タスク τ_l がタスク τ_h に対しプリエンプティブルであるとは，タスク τ_l のジョブ J_l が実行中に τ_h のジョブ J_h がプリエンプトし実行を開始し，その後 J_h の実行が完了した後に J_l が中断した地点から再開し実行を完了した場合のシステムの状態が以下のいずれかの場合の状態と等しいことをいう．

- それぞれのジョブを J_l，J_h の順で逐次的に実行した場合のシステムの状態
- それぞれのジョブを J_h，J_h の順で逐次的に実行した場合のシステムの状態

つまり，PS のようにタスクを実行した結果が，AR モデルで実行した場合の結果か，NP で実行した場合の結果と等しくなることを言う．

タスク τ_l が τ_h に対しプリエンプティブルであることを $\tau_l \triangleleft \tau_h$ と表記する．

4.4 節で行ったタスクの形式化をもとに，タスク τ_l が τ_h に対しプリエンプティブルであるかどうかは以下のように判定できる．

定理 1. それぞれのジョブが外部から受け取る入力が，ジョブに対して定数としてみなせる（ジョブの実行時間，実行順に依存しない）とする．また，タスクのコピーフェーズと書き戻しフェーズはそれぞれアトミックに行われるとする．

タスク τ_l が τ_h に対しプリエンプティブルなのは以下の場合である．

(i) τ_h で更新されるノードが τ_l で参照されない
(ii) τ_l で更新されるノードが τ_h で参照されず，τ_l と τ_h の両方で更新されるノードがない

証明. 4.4 節の形式化より，タスクは，外界とノード更新で参照されるノードの状態から，更新すべき入力ノードと状態ノードの状態への関数としてみなすことができる．つまり，外部からの入力を定数とみなすことができる場合，参照されるノードの状態が同じであればタスクの実行結果も同じである．そのため，タスク τ_l のジョブ J_l とタスク τ_h のジョブ J_h に対し，J_l の実行中に J_h がプリエンプトして実行する場合と，それぞれのジョブを逐次的に実行する場合のノードの状態を比較する．

ここでは，タスク τ_l で更新する入力ノードの集合を I_l，参照するノードの集合を R_l，更新する状態ノードの集合を U_l とする．同様に，タスク τ_h についても I_h, R_h, U_h を定義する．

それぞれのジョブが実行される前のシステムの状態を σ とする．以下の三つの実行について考える．

(a) J_l の実行中に J_h がプリエンプトして実行する．この場合，J_l，J_h はともに σ をもとに実行される．この時の J_l の実行後のシステムの状態を σ_p とする．
(b) それぞれのジョブが J_l，J_h の順で逐次的に実

行する．この場合，まず J_l は σ をもとに実行される．この時の J_l の実行後のシステムの状態を σ_l とする．次に J_h が実行されるが，これは J_l の実行後のシステムの状態 σ_l をもとに実行される．この時の J_h の実行後のシステムの状態を σ_{lh} とする．

(c) それぞれのジョブが J_h，J_l の順で逐次的に実行する．この場合，まず J_h は σ をもとに実行される．この時の J_h の実行後のシステムの状態を σ_h とする．次に J_l が実行されるが，これは J_h の実行後のシステムの状態 σ_h をもとに実行される．この時の J_l の実行後のシステムの状態を σ_{hl} とする．

これらの実行について，

(i) τ_h で更新されるノードが τ_l で更新されない場合，つまり $(R_l) \cap (I_h \cup U_h) = \varnothing$ の場合．

J_l のコピーフェーズで σ から得る R_l の状態と，J_h の実行後に J_l のコピーフェーズで σ_h から得る R_l の状態は等しい．そのため (a) における J_l の結果と (c) における J_l の結果が等しくなる．よって $\sigma_p = \sigma_{hl}$ となり，この場合は τ_l が τ_h に対しプリエンプティブルである．

(ii) τ_l で更新されるノードが τ_h で参照されず，τ_l と τ_h の両方で更新されるノードがない場合，つまり $(R_l) \cap (I_h \cup U_h) = \varnothing$，$(U_l \cup U_h) = \varnothing$ の場合．

まず，J_h のコピーフェーズで σ から得る R_h の状態と，J_l の実行後に J_h のコピーフェーズで σ_l から得る R_h の状態は等しいため，(a) における J_h の結果と (b) における J_h の結果が等しくなる．また，U_l と U_h に共通部分がないため，一方のジョブの更新で他方のジョブの更新を上書きすることがない．よって実行の結果をシステムの状態に書き戻す順番はシステムの最終的な状態に影響を与えない．よって $\sigma_p = \sigma_{lh}$ となり，この場合も τ_l が τ_h に対しプリエンプティブルである．

(iii) τ_l で更新されるノードが τ_h で参照されず，τ_l と τ_h の両方で更新されるノードがある場合．

2 と同様に，(b) における J_h の結果と (c) における J_h の結果は等しい．しかし，U_l と U_h に共通部分があるため，一方のジョブの更新で他方のジョブの更新を上書きする．よって実行の結果をシステムの状態に書き戻す順番はシステムの最終的な状態に影響を与えるため $\sigma_p = \sigma_{lh}$ とはならず，この場合は τ_l が τ_h に対しプリエンプティブルでない．

(iv) τ_h で更新されるノードが τ_l で参照され，τ_l で更新されるノードが τ_h で参照される場合．

この場合は J_h のコピーフェーズで σ から得る R_h の状態と J_l のコピーフェーズで σ から得る R_l の状態が等しくなく，また J_l のコピーフェーズで σ から得る R_l の状態と J_h のコピーフェーズで σ から得る R_l の状態も等しくないため，(a) と (b)，(c) の実行の結果は異なる．よってこの場合も τ_l が τ_h に対しプリエンプティブルでない．

□

5.3 スケジューリング方式

前節で述べたプリエンプティビリティを用いて，FreacTime におけるスケジューリング方式を以下のように提案する．

- イベントが発生した時，そのイベントに対応するタスクを優先度付きキューに追加する．
- 優先度付きキューの先頭にタスク τ_h が追加された時，
  - 現在実行中のタスクの優先度が追加されたタスク τ_h の優先度より低く，実行中のタスクと中断されているタスクの全てが τ_h に対してプリエンプティブルである場合，実行中のタスクを中断し，タスク τ_h を実行する．
  - それ以外の場合，現在実行中のタスクを継続する．
- 実行中のタスクが終了した時，優先度付きキューの先頭にタスク τ_h がある場合，
  - 実行を中断しているタスクがない場合はタスク τ_h を実行する．
  - 実行を中断しているタスクがある場合，そのうちのもっとも優先度が高いタスクを τ_i とすると，

* タスク τ_i が τ_h よりも優先度が低く，中断しているタスク全てが τ_h に対してプリエンプティブルである場合，タスク τ_h の実行を開始する．
* それ以外の場合，タスク τ_i の実行を再開する．

5.4 スケジューラビリティ解析

スケジューラビリティ解析は，システムを構成するすべてのタスクがデッドラインを満たすかどうかを判定するものである．つまり，タスクがリリースされた時刻から実行が完了するまでの時刻の最悪の場合（最悪応答時間）を考え，これがデッドラインを超えないかどうかを判定する．本節では，前節までで述べたスケジューリング方式におけるスケジューラビリティ解析を行う．

提案したスケジューリング方式は，特定の条件を満たす場合はプリエンプションを行い，それをを満たさない場合はプリエンプションを行わないというものである．そのため，PS と NP の中間的なスケジューリング方式であると言える．そのような PS と NP の中間的なスケジューリング方式は，Limited Preemptive Scheduling として研究が行われている [8]．提案したスケジューリング方式におけるスケジューラビリティ解析は，この Limited Preemptive Scheduling のスケジューラビリティ解析の結果を参考に行う．

スケジューラビリティ解析は，最悪応答時間を求めることで行われる．最悪応答時間は，高い優先度のタスクの実行などによって対象としているタスクの実行がもっとも遅れるような状況を考えることで求められる．そのような状況を Critical Instant と呼ぶ．PS における Critical Instant は，対象となるタスクとそれよりも高い優先度のタスクすべてが同時にリリースされるような状況である．最悪応答時間の計算は，この Critiacl Instant のもとで応答時間を求めることで行われる．

一方，低優先度のタスクが高優先度のタスクにプリエンプトされない場合がある時は他にも考慮すべき概念がある．そのうちの一つは Blocking Time である．これは，対象とするタスクがリリースされる前に，より低優先度のタスクが実行を開始しており，そのタスクにプリエンプトできずに待機している時間である．そしてもう一つは Self-pushing Phenomenon である．これは，対象とするタスクの実行によってより高優先度のタスクの実行の開始が遅れ，それが最終的に対象とするタスクの実行を遅らせる現象である．

このような現象があるため，スケジューラビリティ解析は Level-i Active Period [6] という時間区間で考える必要がある．Leval-i Active Period とは区間 $[a, b)$ であり，$t \in (a, b)$ となるすべての時刻 t において優先度 $\mathbf{P}_i$ 以上のタスクで実行すべき状態にあるものが存在し，$t = a$ と $t = b$ において優先度 $\mathbf{P}_i$ 以上のタスクで実行すべき状態にあるものがないような区間である．

ここまでで述べた概念を用いて，タスク τ_i の最悪応答時間 $\mathbf{R}_i$ を求める．

まず Blocking Time B_i を求める．これは，τ_i より優先度の低く τ_i がプリエンプトできないタスクが，τ_i のリリースの直前に実行を開始した場合を考えれば良いので，

$$B_i = \max\{\mathbf{C}_j - 1 \mid \exists h.(\tau_h \not\prec \tau_i \wedge \mathbf{P}_h \leq \mathbf{P}_j < \mathbf{P}_i)\}. \tag{1}$$

Level-i Active Period L_i は，B_i に加え，L_i のうちにリリースされたすべてのタスクが完了するまでの時間となる．よって

$$L_i = B_i + \sum_{h.\mathbf{P}_h \geq \mathbf{P}_i} \left\lceil \frac{L_i}{\mathbf{T}_h} \right\rceil \mathbf{C}_h. \tag{2}$$

となる．よって，τ_i の最悪応答時間 $\mathbf{R}_i$ は，

$$K_i = \left\lceil \frac{L_i}{\mathbf{C}_i} \right\rceil \tag{3}$$

で表される K_i 個のジョブについて応答時間を求めれば良い．

k 番目のジョブの開始時刻 $s_{i,k}$ は，B_i に加え，$k-1$ 個のジョブが完了するまでの時間と，$s_{i,k}$ までにリリースされる τ_i より高優先度のタスクの実行時間を考慮することで求められる．よって，

$$s_{i,k} = B_i + (k-1)\mathbf{C}_i + \sum_{h.\mathbf{P}_h > \mathbf{P}_i} \left(\left\lfloor \frac{s_{i,k}}{\mathbf{T}_h} \right\rfloor + 1 \right) \mathbf{C}_h. \tag{4}$$

k 番目のジョブの終了時刻 $f_{i,k}$ は，$s_{i,k}$ に $\mathbf{C}_i$ を加え，さらに k 番目のジョブに対しプリエンプトして

実行されるタスクの実行時間を考えれば良いので，

$$f_{i,k} = s_{i,k} + \mathbf{C}_i + \sum_{h.\mathbf{P}_h > \mathbf{P}_i \wedge \tau_i \lhd \tau_h} \left(\left\lceil \frac{f_{i,k}}{\mathbf{T}_h} \right\rceil - \left(\left\lfloor \frac{s_{i,k}}{T_h} \right\rfloor + 1 \right) \right) \mathbf{C}_h \tag{5}$$

となる．

最後に，k 番目のジョブの最悪応答時間はジョブの終了時刻からリリース時刻を引いたものとなるので，K_i 個のジョブについてそれらの最大値を取ることで，

$$\mathbf{R}_i = \max_{k \in [1, K_i]} \{ f_{i,k} - (k-1)\mathbf{T}_i \} \tag{6}$$

と求められる．

よって，システムがスケジューリング可能であるための条件は，これまでで求めた最悪応答時間を用いて

$$\forall i = 1, \ldots, n. \mathbf{R}_i \leq \mathbf{D}_i \tag{7}$$

となる．

PS でスケジューリング可能な場合は Self-pushing Phenomenon が発生しない [17] ため最初のジョブのみ考えれば良いので，最悪応答時間の計算は

$$\mathbf{R}_i = s_{i,1} + \mathbf{C}_i + \sum_{h.\mathbf{P}_h > \mathbf{P}_i \wedge \tau_i \lhd \tau_h} \left(\left\lceil \frac{\mathbf{R}_i}{\mathbf{T}_h} \right\rceil - \left(\left\lfloor \frac{s_{i,1}}{\mathbf{T}_h} \right\rfloor + 1 \right) \right) \mathbf{C}_h \tag{8}$$

と簡略化できる．

6 関連研究

6.1 同期的データフロー言語

SIGNAL [4]，Lustre [11] は同期的プログラミング言語であり，リアクティブシステムの記述に用いられる．これらの言語では，クロックと呼ばれる値を明示的に扱うことで時間に関する記述を可能とする．

SCADE [9] は，Lustre や同じく同期的言語である Esterel [5] などを組み合わせて作られたプログラミング言語である．この言語はセーフティクリティカルな組込みシステム向けの言語であり，航空機や原子力発電所などの制御システムの開発に用いられている．

6.2 P-FRP における実行モデル

P-FRP では，前述の Abort-and-Restart（AR）モデルにおける優先度割り当てやスケジューラビリティテストなど研究が行われている [19]．また，AR モデル以外の実行モデルの提案もいくつか行われている．Deferred Abort（DA）モデル [16] はそのうちの一つで，タスクを二つの領域に分け，前半は AR モデルで実行し，後半はノンプリエンプティブスケジューリングで実行するというものである．また，Deferred Start（DS）モデル [18] も P-FRP の実行モデルの一つである．これは AR モデルにおける Non-work-conserving なモデルで，タスクの実行開始時刻を遅らせて優先度の高いタスクの到着を待つことでプリエンプションの回数を減らしている．

本研究では，タスク同士の独立性に注目し，プリエンプトされたタスクが再開時に最初からやり直す必要がない場合にプリエンプションを行うことでタスクのやり直しをなくすスケジューリング方式を提案している．

6.3 Limited Preemptive Scheduling

Limited Preemptive Scheduling [8] は，プリエンプティブスケジューリングとノンプリエンプティブスケジューリングの中間的なスケジューリング方式である．具体的なスケジューリング方式として，Preemption Thresholds Scheduling（PTS） [15]，Deferred Preemption Scheduling（DPS） [3]，Fixed Preemption Point（FPP） [7] の 3 つが提案されている．

本研究で提案したスケジューリング方式もプリエンプティブスケジューリングとノンプリエンプティブスケジューリングを組み合わせたものであり，Limited Preemptive Scheduling と類似している．一方で，提案したスケジューリング方式は FRP が持つ制約を満たすために設計されたものだが，Limited Preemptive Scheduling はプリエンプションによるオーバーヘッドを抑えるなどプリエンプティブスケジューリングより効率的なスケジューリングを目指している，という点で相違する．

7 結論と今後の課題

本研究では，FRP によるリアルタイムシステムの記述言語として，FreacTime では，個々のタスクとそのデッドラインを明示的に記述することができる．また，ノードに対しタイミングと呼ばれる抽象的な時刻情報を持たせることで，タスクにおいてどのノード

更新処理が行われるかを簡潔に表現することができる．さらに，ノードの関係をもとにしたタスク間の独立性を定義し，それを用いたスケジューリング方式を提案した．そして提案したスケジューリング方式に対するスケジューラビリティ解析を示した．

今後の課題としては，提案したスケジューリング方式の評価が挙げられる．P-FRP では，Abort-and-Restart モデルが提案されており，その他の実行モデルに対するスケジューリング方式の提案も行われている．それらに対して本研究で提案したスケジューリング方式との比較を行い，提案したスケジューリング方式の有効性を調べる必要がある．また，FreacTime の処理系の実装を行い，より実社会を意識したケーススタディへの適用を行うことで，提案手法の有用性を示すことも今後の課題である．

謝辞 本研究の一部は JSPS 科研費 JP22K11967 および JP24K14892 の助成を受けている．

参 考 文 献

[1] Akesson, B., Nasri, M., Nelissen, G., Altmeyer, S., and Davis, R. I.: A comprehensive survey of industry practice in real-time systems, *Real-Time Systems*, Vol. 58, No. 3(2022), pp. 358–398.

[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(2013), pp. 52:1–52:34.

[3] Baruah, S.: The limited-preemption uniprocessor scheduling of sporadic task systems, *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, 2005, pp. 137–144.

[4] Benveniste, A., Le Guernic, P., and Jacquemot, C.: Synchronous programming with events and relations: the SIGNAL language and its semantics, *Science of Computer Programming*, Vol. 16, No. 2(1991), pp. 103–149.

[5] Berry, G. and Gonthier, G.: The Esterel synchronous programming language: design, semantics, implementation, *Science of Computer Programming*, Vol. 19, No. 2(1992), pp. 87–152.

[6] Bril, R. J., Lukkien, J. J., and Verhaegh, W. F.: Worst-Case Response Time Analysis of Real-Time Tasks under Fixed-Priority Scheduling with Deferred Preemption Revisited, *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, 2007, pp. 269–279.

[7] Burns, A.: Preemptive Priority Based Scheduling: An Appropriate Engineering Approach, *Advances in Real-Time Systems*, (1994), pp. 225–248.

[8] Buttazzo, G. C., Bertogna, M., and Yao, G.: Limited Preemptive Scheduling for Real-Time Systems. A Survey, *IEEE Transactions on Industrial Informatics*, Vol. 9, No. 1(2013), pp. 3–15.

[9] Colaço, J.-L., Pagano, B., and Pouzet, M.: SCADE 6: A formal language for embedded critical software development (invited paper), *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*, 2017, pp. 1–11.

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

[11] 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.

[12] Kaiabachev, R., Taha, W., and Zhu, A.: E-FRP with Priorities, *7th ACM/IEEE International Conference on Embedded Software (EMSOFT 2007)*, 2007, pp. 221–230.

[13] Sawada, K. and Watanabe, T.: Emfrp: A Functional Reactive Programming Language for Small-Scale Embedded Systems, *MODULARITY Companion 2016: Companion Proceedings of the 15th International Conference on Modularity*, ACM, Mar. 2016, pp. 36–44.

[14] Sogo, K., Tsuji, Y., Moriguchi, S., and Watanabe, T.: Periodic and Aperiodic Task Description Mechanisms in an FRP Language for Small-Scale Embedded Systems, *Proceedings of the 10th ACM SIGPLAN International Workshop on Reactive and Event-Based Languages and Systems*, REBLS 2023, New York, NY, USA, ACM, 2023, pp. 43–53.

[15] Wang, Y. and Saksena, M.: Scheduling fixed-priority tasks with preemption threshold, *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications. RTCSA'99 (Cat. No.PR00306)*, 1999, pp. 328–335.

[16] Wong, H. and Burns, A.: Priority-Based Functional Reactive Programming (P-FRP) Using Deferred Abort, *2015 IEEE 21st International Conference on Embedded and Real-Time Computing Systems and Applications*, 2015, pp. 227–236.

[17] Yao, G., Buttazzo, G., and Bertogna, M.: Feasibility Analysis under Fixed Priority Scheduling with Fixed Preemption Points, *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, 2010, pp. 71–80.

[18] Zou, X., Cheng, A. M. K., and Jiang, Y.: A Non-Work-Conserving Model for P-FRP Fixed Priority Scheduling, *2016 13th International Conference on Embedded Software and Systems (ICESS)*, 2016, pp. 12–17.

[19] Zou, X., Cheng, A. M., and Jiang, Y.: P-FRP task scheduling: A survey, *2016 1st CPSWeek Workshop on Declarative Cyber-Physical Systems*

(DCPS), IEEE, 2016, pp. 1–8.