情報処理学会論文誌　プログラミング　Vol.6　No.3　1–19 (Dec. 2013)

動的再構成可能システムの仕様記述言語の提案およびその検証実験

山田 英史[1,a)]　中居 祐輝[1]　山根 智[1,b)]

受付日 2013年1月29日, 採録日 2013年4月24日

概要：近年の組込み機器の多機能化を受け，消費電力の抑制や小型化のために動的再構成可能プロセッサ（DRP）が注目されている．本稿では，CPU と DRP から構成される動的再構成可能システムを対象とした仕様記述言語を提案する．提案する仕様記述言語では，線形階層ハイブリッドオートマトンとオブジェクト指向を組み合わせることで，システム構成の動的変化を表現している．また，提案言語を対象とした検証器を開発し，その有効性を実証する．

キーワード：モデル検査，動的再構成可能プロセッサ（DRP），ハイブリッドオートマトン

Proposal of Specification Language and Verification Experiment for Dynamically Reconfigurable System

Hidefumi Yamada[1,a)]　Yuki Nakai[1]　Satoshi Yamane[1,b)]

Received: January 29, 2013, Accepted: April 24, 2013

Abstract: Recently, embedded systems begin to have various functions, Dynamically Reconfigurable Processor (DRP) draws attention to solution for the miniaturization and saving energy. In this paper, we propose the specification language for Dynamically Reconfigurable System, which is composed of CPU and DRP. The proposed language describes dynamically changing the system using Hierarchical Linear Hybrid Automaton and Object Orientation. Then, we develop the verifier for the proposed specification language, and verify the system.

Keywords: model checking, Dynamically Reconfigurable Processor (DRP), Hybrid Automata

1. 序論

1.1 背景

近年，組込み機器は様々な機能を持つようになり，専用処理用プロセッサ数の増加が小型化や省電力化の障害となっている．従来，省電力化を図る場合，高速・高負荷の処理向けに ASIC や DSP などの専用プロセッサを用意してきた．しかし，多機能化が進むにつれて用意すべき専用プロセッサの数が増え，消費電力の増加が無視できなくなっている．また，システムの複雑化が進み，システムの安定した動作を保証する手法も必要となっている．

1.2 動的再構成可能プロセッサ

省電力化と小型化を両立させる方法の 1 つとして，近年，動的再構成可能プロセッサ [17] が注目されている．動的再構成可能プロセッサ（Dynamically Reconfigurable Processor: DRP）は粗粒度構成のプログラマブルなプロセッサである．DRP の代表的な特徴として，次のことがあげられる．

(1) 機能を複数のコンテキスト（論理回路構成）に分解し，コンテキストを切り替えながら少ない面積で処理を行う [19].
(2) コンテキストごとに，プロセッサの動作周波数を変更する [14].

[1] 金沢大学
Kanazawa University, Kanazawa, Ishikawa 920–1192, Japan
[a)] yamada@csl.ec.t.kanazawa-u.ac.jp
[b)] syamane@is.t.kanazawa-u.ac.jp

(3) 同一基盤上で同時に異なる複数の処理を実行する [13].
(4) コンテキストサイズを最適化することにより，電力効率の最適化を図れる [18].

DRP は柔軟な処理構成の変更や小型化に適したプロセッサである反面，動作周波数の変更や並列実行されるタスク数が変更されるなど，挙動や応答時間を予測しにくい点がある．そのため，DRP を搭載したシステムの安全性を保証する手段が必要とされる．

1.3 仕様記述

複雑化するシステムの安定性を保証するには，システム全体を把握するために，正確に仕様を記述することが重要となる．仕様記述言語として，一般的には UML が利用されている．UML は様々な要素を持つ反面，あらゆるシステムをターゲットとしているために言語体系が複雑であり，特定のシステムへの適用を困難にしている．さらに，成果物に対する検証が難しく，成果物の正当性が保証しにくい．また，DRP のように動作周波数が動的に変更されるハイブリッドシステムの場合，従来の CPU によるリアルタイムシステムよりも仕様記述が複雑となる．特に，DRP は複数の処理を同時実行可能であり，動作周波数の変更，システムの構成変更など，既存の仕様記述言語では記述困難であるという特徴を持つ．この特徴を記述するには，構成変更を記述できる柔軟な仕様記述言語が必要である．

1.4 形式的検証

システムの安全性を保証する手段として，テストやシミュレーションが一般的に利用されている．しかし，テストやシミュレーションはシステムを部分的にしか検証できず，システム全体の安全性を保証するのは困難である．特に，DRP は回路構成の変更や周波数の変更などにより挙動が予測しにくく，また回路やコンテキストなどのリソースに起因する問題も発生する．そのため，テストやシミュレーションによって DRP の安全性を保証するのは困難である．

これを解決する手法として形式的検証がある．形式的検証は，システム全体をモデル化し，そのモデルを網羅的に検証することでシステム全体の安全性を保証する手法である．形式的検証の網羅性により，予測困難な挙動をする DRP でも安全性検証が可能となる．

1.5 目的

本稿では，CPU と DRP から構成される動的再構成可能システムを対象とした仕様記述言語を提案する．仕様記述言語に必要な要件として，次の点があげられる．

(1) システム全体の動的な構成変化
(2) DRP の動作周波数の変化
(3) 同期通信
(4) CPU-DRP 間のデータ通信

これらの要件を部分的に満たす言語は存在するが，すべてを満たす言語は存在しない．本稿では，これら要件をすべて満たす仕様記述言語を提案する．また，提案した仕様記述言語を用いてシステムの仕様記述を行い，システムの正当性を形式的検証によって実証する．

1.6 関連研究

動的再構成可能システムの仕様記述言語としては，リアクティブモデル，リアルタイムモデル，ハイブリッドモデルのそれぞれのモデルがあり，仕様記述のスタイルとして，プロセス代数，オートマトン，ペトリネットがあげられる．たとえば，Deshpande らの SHIFT [6] と Kratz らの R-Charon [10] はハイブリッドオートマトン [1] を基礎とする仕様記述言語である．これらは，ハイブリッドネットワークの構成要素間の通信リンクを動的に変化させることで構造の動的変化を表現する記述言語であり，バイオケミカルプロセスや輸送システム，またモジュラ再構成可能ロボットなどを対象としている．しかし，これらの言語はイベント駆動の処理が記述できず，本稿で扱う動的再構成可能システムに不向きである．また，Attie らはリアクティブオートマトンである Input/Output Automata を拡張し，動的なオートマトンの生成・消滅に対応させた仕様記述言語を開発している [4]．DRP のモデル化に関する研究として，Teich らや Onogi らがリアクティブオートマトンを用いて離散モデル化した例がある [11], [12]．CPU と DRP から構成される動的再構成可能システムを扱った例としては，南らのハイブリッドオートマトンを使用した例がある [20]．南らの実験では，CPU と DRP を分割して検証することにより，悲観的なスケジューラビリティ検証をしている．

本稿で提案する仕様記述言語では，階層時間オートマトン [16] を拡張した線形階層ハイブリッドオートマトンを提案し，また，オブジェクト指向を取り入れることにより構造の動的変化を記述できるようにする．そして，同期通信やデータ通信の記述手法を導入し，各オートマトン間での協調動作を表現可能とする．さらに，操作的意味論を定義することにより仕様記述言語の形式的検証を可能にする．また，計算機上に検証器を実装し，仕様記述言語によるシステムの検証を実際に行う．システムの検証性質として，DRP 上の回路面積が足りず再構成ができない場合のスケジューラビリティを扱う．これは動的再構成システム特有の性質であり，検証実験を通して，ハードウェアとソフトウェアの制約を組み合わせた検証が可能であることを示す．これらにより，本稿で提案する仕様記述言語が実際のシステムに適用可能であることを示す．

1.7　本稿の構成

まず，2 章で仕様記述言語の定義を行い，3 章で提案言語による動的再構成可能システムの仕様記述を行う．また，4 章では仕様記述したモデルの検証実験を行い，5 章でまとめと今後の課題について述べる．

2. 仕様記述言語

この章では，仕様記述言語を定義する．仕様記述言語は，動的再構成可能システムの構文と意味，また動的再構成可能システムの安全性を検証する到達可能性問題の定義から構成される．到達可能性問題は，動的再構成可能システムの意味にあたる遷移系の上で定義される．

2.1　概要

動的再構成可能システムでは，CPU 上で CPU-Controller がタスク（Task）を制御しており，CPU は DRP と相互通信することによってイベント同期やデータの送受信を行う．また，DRP 上では DRP-Controller によって協調タスク（CoTask）が制御される．本稿では，CPU と DRP が協調動作する動的再構成可能システムを仕様記述するために，次の手法を用いる．

(1) 動的な構成変化を表現するために，オブジェクト指向を用いる．オブジェクト指向を表現できる既存の言語には R-Charon [10] がある．
(2) 振舞いの階層構造を表現するために，階層時間オートマトン [16] を用いる．また，階層構造の状態を保持するために，履歴を用いる．
(3) 周波数の変化を表現するために，ハイブリッドオートマトン [1] を用いる．なお，本稿で提案する仕様記述言語では，周波数の変化を定数に限定した線形ハイブリッドオートマトンを採用する．
(4) CPU と DRP 間のイベント同期を表現するために，イベント同期を用いる．イベント同期が表現できる既存の言語にはハイブリッドオートマトン [1] やステートチャートがある．
(5) CPU と DRP 間のデータ送受信を表現するために，Hoare の CSP [9] を用いる．

本稿では，上記の手法を組み合わせることにより，動的再構成可能システムの仕様記述言語を実現する．なお，本稿で提案する仕様記述言語では，オブジェクトの無限生成やイベント同期による無限遷移などの *live lock* が発生する恐れがある．このような現象は活性の問題として扱う必要があり，本稿の範囲を外れるため，本稿では *live lock* が発生するモデルは扱わないとする．

2.2　準備

まず，動的再構成可能システムを定義するうえで必要となる変数やデータ構造，制約式と更新式，アクションについて定義する．また，本稿では次の表記を使用する．

- $[\![a]\!]$ は a の意味論を表す．
- 集合 $\mathcal{A}$，$\mathcal{B}$ に対して，$\mathcal{A}$ から $\mathcal{B}$ への写像全体の集合を $\mathcal{B}^{\mathcal{A}} = \{f \mid f : \mathcal{A} \to \mathcal{B}\}$ と記述する．
- $[a_0, a_1, \cdots]$ は順序付き集合（リスト）を表す．リストでは要素の重複を認めるとする．また $[a_0, a_1, \cdots]_i$ はリストの i 番目の要素を表す．
- リスト $list_1$ とリスト $list_2$ を連結したリストを $list_1 :: list_2$ と記述する．

2.2.1　時間変数と離散変数

動的再構成可能システムを表すうえで，時間を表す変数，離散値を表す変数，動的構造を表す変数，順序列を表す変数の 4 つが必要となる．本稿では，それぞれを表現する変数としてクロック変数，離散変数，参照変数およびリスト変数を定義する．この項では，クロック変数と離散変数を定義する．

定義 2.1（クロック変数とクロック評価）
X_c を，非負の実数値を持つクロック変数の有限集合とする．このとき，$\nu_c : X_c \to \mathbb{R}_{\geq 0}$ を任意のクロック変数 x_c に非負の実数を割り当てる関数と定義し，これをクロック評価と呼ぶ．クロック評価 $(\nu_c + \delta)$ は各クロック変数の評価値に $\delta \in \mathbb{R}_{\geq 0}$ を足した値を表す．また，$\nu_c[X'_c := 0]$ によって，集合 $X'_c \subseteq X_c$ の要素のクロック変数の評価値を 0 に，それ以外については ν_c と同じ評価値を割り当てることを表す．

□

定義 2.2（離散変数と離散変数評価）
X_d を，整数値を持つ離散変数の有限集合とする．このとき，$\nu_d : X_d \to \mathbb{Z}$ を任意の離散変数 x_d に整数値を割り当てる関数と定義し，これを離散変数評価と呼ぶ．また，$\nu_d[X'_d := z]$ によって，集合 $X'_d \subseteq X_d$ の要素の離散変数の評価値を $z \in \mathbb{Z}$ に，それ以外については ν_d と同じ評価値を割り当てることを表す．

□

2.2.2　クラス/オブジェクトと参照変数

システムを再構成する単位として，本稿ではクラスとオブジェクトを用いる．クラスはシステムの各モジュールの設計を表し，オブジェクトは稼働中のモジュールを表す．クラス/オブジェクトの定義については 2.3.2 項に後述する．ここでは，クラス/オブジェクトと連携してシステムの動的な構造変化を表現する参照変数と，その操作を定義する．

定義 2.3（参照変数と参照変数評価）
X_r を，オブジェクトを示すことができる参照変数 x_r の有限集合とする．このとき，$\nu_r : X_r \to \mathit{Objects} \cup \{\mathit{null}\}$ を任意の参照変数 x_r にオブジェクトを割り当てる関数と定義し，これを参照変数評価と呼ぶ．また，$\nu_r[X'_r := \mathit{obj}^*]$ によって，集合 $X'_r \subseteq X_r$ の要素の参照変数の評価値を $\mathit{obj}^* \in \mathit{Objects} \cup \{\mathit{null}\}$ に，それ以外については ν_r と同

じ評価値を割り当てることを表す．

□

ここで，*null* は参照変数がオブジェクトを示していないことを表す定数であり，これを無効参照と呼ぶ．

クラスとオブジェクトは，振舞いを表現するオートマトン（2.3.1 項）の集合と複数個の変数から構成される．参照変数では，参照変数が示すオブジェクトが保持する変数へのアクセスを表現できる．

定義 2.4（参照変数による変数へのアクセス）
参照変数 x_r がオブジェクト $obj \in Objects$ を示している場合，$x_r.y$ は，obj が持つ変数 y と等価である．□

次に，クラスをオブジェクトに変換する関数 create と，参照変数が保持するオブジェクトを破棄する関数 destroy を定義する．

定義 2.5（オブジェクトの生成）
関数 create は，動的クラス cls_{dyn} を動的オブジェクト obj_{dyn} に変換する関数である．

$$\text{create} : cls_{dyn} \mapsto obj_{dyn}$$

ただし，create は呼ばれるたびに異なる動的オブジェクトを生成する．そのため，同じ動的クラスから異なる動的オブジェクトが生成されることに注意が必要である．

□

定義 2.6（オブジェクトの破棄）
関数 destroy は，参照変数 x_r が示す動的オブジェクトを破棄し，x_r に *null* を割り当てる関数である．また，この関数によって破棄された動的オブジェクトは，再構成されたシステムに含まれない．

□

また，オブジェクトは変数 *self* で自分自身を参照することができる．

定義 2.7（オブジェクトの自己参照）
変数 $self \in X_r$ は，それを呼び出したオブジェクトの参照を示す．ただし，*self* は呼び出すオブジェクトによって示す参照が変わり，また，代入不可能である．

□

本稿で提案する記述言語では，参照変数とクラス/オブジェクト，また create/destroy 関数を用いてシステムの再構成を表現する．クラスおよびオブジェクトについては 2.3.2 項に後述する．

2.2.3　リスト変数

リスト変数は，順序付き集合（リスト）を保持する変数のことである．

定義 2.8（リスト変数とリスト変数評価）
X_l を，リストを示すことができるリスト変数 x_l の有限集合とする．このとき，$\nu_l : x_l \mapsto [a_0, a_1, \cdots]$ を任意のリスト変数 x_l にリストを割り当てる関数と定義し，これをリスト変数評価と呼ぶ．また，$\nu_l[X'_l := list]$ によって，集合 $X'_l \subseteq X_l$ の要素のリスト変数の評価値を *list* に，それ以外については ν_l と同じ評価値を割り当てることを表す．

□

リスト変数を操作する関数を次に定義する．

定義 2.9（リスト変数の操作）
リスト変数の関数として，次の関数を定義する．

- （先頭取得）shift: $X_l \to X$
  リスト変数 $x_l = [a_0, a_1, \cdots, a_n]$ について，変数 a_0 を返し，x_l の評価値を $[a_1, \cdots, a_n]$ に更新する．
- （先頭追加）unshift: $X_l \times X$
  リスト変数 $x_l = [a_0, a_1, \cdots, a_n]$ と変数 y を受け取り，x_l の評価値を $[y, a_0, a_1, \cdots, a_n]$ に更新する．
- （末尾取得）pop: $X_l \to X$
  リスト変数 $x_l = [a_0, a_1, \cdots, a_n]$ について，変数 a_n を返し，x_l の評価値を $[a_0, a_1, \cdots, a_{n-1}]$ に更新する．
- （末尾追加）push: $X_l \times X$
  リスト変数 $x_l = [a_0, a_1, \cdots, a_n]$ と変数 y を受け取り，x_l の評価値を $[a_0, a_1, \cdots, a_n, y]$ に更新する．
- （存在検査）empty: $X_l \to \{true, false\}$
  リスト変数 x_l を受け取り，リスト変数に要素があれば *false*，1 つもない場合は *true* を返す．

□

2.2.4　変数の制約と更新式

ここまで定義してきた 4 つの変数の全体集合 $X = X_c \cup X_d \cup X_r \cup X_l$ を，変数集合と呼ぶ．ここでは，変数集合 X に対する制約（一次不等式，*flow* 条件）と更新式を定義する．

定義 2.10（変数の一次不等式）
変数集合 $X = X_c \cup X_d \cup X_r \cup X_l$ に対する一次不等式 ϕ を以下の構文で定義する．

$$\begin{aligned}\phi ::= {} & asap \mid true \mid false \mid \gamma_1 \sim \gamma_2 \mid \phi_1 \wedge \phi_2 \\ \mid {} & x_r = null \mid x_r \neq null \\ \mid {} & \text{empty}(x_l) = true \mid \text{empty}(x_l) = false\end{aligned}$$

ただし，

$$\gamma ::= x_c \mid x_d \mid z \mid \gamma_1 + \gamma_2 \mid \gamma_1 - \gamma_2$$

であり，$\sim \in \{<, \leq, =, \geq, >\}$，$z \in \mathbb{Z}$，$x_c \in X_c$，$x_d \in X_d$，$x_r \in X_r$，$x_l \in X_l$ である．また，不等式の意味 $[\![\phi]\!]$ は不等式を充足する評価値 ν の集合である．ここで，*asap* は時間遷移よりも離散遷移が優先される制約を示す．*asap* の動作については 2.4.3 項に後述する．変数の一次不等式 ϕ の集合を $Guard(X)$ とする．□

定義 2.11（クロック変数の *flow* 条件）
クロック変数集合 X_c に対して，その増加量を与える *flow* 条件を以下の構文で定義する．

$$f ::= \dot{x}_c := z_{\geq 0} \mid \dot{x}_c := \dot{y}_c \mid f_1 \,,\, f_2$$

ここで，$\dot{x}_c$ はクロック変数 x_c の導関数 $\frac{dx_c}{dt}$ のことである．*flow* 条件はカンマで並べることができ，また任意の変数の傾きを別の変数の傾きとして設定することもできる．ただし，$z_{\geq 0} \in \mathbb{Z}_{\geq 0}$ である．*flow* 条件の意味 $[\![f]\!]$ は，各クロック変数 x_c に傾き $z_{\geq 0}$ を割り当てる関数である．*flow* 条件の集合を $Flow(X_c)$ とする．

□

定義 2.12（変数の更新式）
変数集合 $X = X_c \cup X_d \cup X_r \cup X_l$ に対する更新式 u を以下の構文で定義する．

$$
\begin{aligned}
u ::= \; & x_c := 0 \\
\mid \; & x_d := u_d \; (\because u_d ::= z \mid x_d \mid u_{d1} + u_{d2} \mid u_{d1} - u_{d2}) \\
\mid \; & x_r := \mathrm{create}(cls_{dyn}) \mid \mathrm{destroy}(x_r) \\
\mid \; & \mathrm{push}(x_l, x) \mid \mathrm{unshift}(x_l, x) \\
\mid \; & x := \mathrm{pop}(x_l) \mid x := \mathrm{shift}(x_l)
\end{aligned}
$$

ただし，$z \in \mathbb{Z}$，$x_c \in X_c$，$x_d \in X_d$，$x_r \in X_r$，$x_l \in X_l$ である．また，$x \in X$ はリスト内要素と同じ型の任意の変数である．変数の更新式 u のリストを *up*，また *up* の集合を $Update(X)$ とする．

□

2.2.5　アクション

システムが複数のモジュールの連携で表現される場合，それらのモジュールの同期を表現する必要がある．ここでは，同期ポイントとして入力/出力アクションを定義し，それらアクションの対応も定義する．

定義 2.13（入力アクション）
入力アクション act_{in} を以下の構文で定義する．

$$act_{in} ::= act\,?\,x_v$$

ここで，act はアクション名，$x_v \in X_d \cup X_r$ である．また，入力アクションの意味 $[\![act_{in}]\!]$ はアクション名と変数の組 (act, x_v) である．入力アクションの全体集合を Act_{in} とする．

□

定義 2.14（出力アクション）
出力アクション act_{out} を以下の構文で定義する．

$$act_{out} ::= act\,!\,val \mid act\,!\,x_v$$

ここで，act はアクション名，$val \in \mathbb{Z} \cup Objects$，$x_v \in X_d \cup X_r$ である．また，出力アクションの意味 $[\![act_{out}]\!]$ はアクション名と値の組 (act, val) である．ただし，出力アクションに変数 x_v が指定されていた場合は，同期するときの変数の評価値 $\nu(x_v)$ を値 val とする．出力アクションの全体集合を Act_{out} とする．

□

定義 2.15（入出力アクションの対応）
出力アクション $[\![act_{out}]\!] = (a_{out}, val)$ は，$a_{in} = a_{out}$ となる入力アクション $[\![act_{in}]\!] = (a_{in}, x_v)$ のすべてに対応する．

□

出力アクションは，同時に複数の入力アクションへ対応する．そのため，出力アクションはブロードキャスト通信と見なすことができる．また，アクションは参照変数を通すことにより，別のオブジェクトとも同期ができる．

定義 2.16（参照変数によるアクション同期）
参照変数 x_r がオブジェクト *obj* を示している場合，$x_r.act$ は，*obj* が持つアクション *act* と等価である．

□

参照変数を通して同期をさせることにより，外部モジュールとの同期とモジュール内部での同期を書き分けることが可能となる．

2.3　構文

ここでは，動的再構成可能システムの構文を定義する．動的再構成可能システムの構文は 3 段階に定義される．

- モジュール内部の処理を表現する “線形階層ハイブリッドオートマトン”
- モジュールの構成を表現する “クラス” および “オブジェクト”
- システム全体の構造を表現する “システム”

2.3.1　線形階層ハイブリッドオートマトンの構文

線形階層ハイブリッドオートマトン（Hierarchical Linear Hybrid Automaton: HLHA）は，モジュール内部の各処理の挙動を表現する．線形階層ハイブリッドオートマトンの構文を次に定義する．

定義 2.17（線形階層ハイブリッドオートマトンの構文）
線形階層ハイブリッドオートマトンの構文 $HLHA = \langle L, h, type, L_0, X, I, \Gamma, Act, Hist, T \rangle$ は，次の要素からなる．

- ロケーションの有限集合 L
- 階層関数 h: $L \to 2^L$
  $l \in L$ のサブロケーションを返す関数．
- 型関数 *type*: $L \to \{BASIC,\ OR\}$
  $l \in L$ の型を返す関数．型 $BASIC$ を持つロケーションは基底ロケーションであり，$h(l) = \emptyset$ となる．型 OR を持つロケーションは OR ロケーションであり，$h(l) \neq \emptyset$ となる，つまりサブロケーションを持つロケーションである．
- 初期ロケーション集合 $L_0 \subseteq L$
  特に最上位のロケーションを $root \in L_0$ とする．
- 変数の有限集合 $X = X_c \cup X_d \cup X_r \cup X_l$
  ただし，X_c はクロック変数，X_d は離散変数，X_r は参照変数，X_l はリスト変数の有限集合である．
- 基底ロケーションに不変条件を割り当てる関数 I: $L \to Guard(X)$

- 基底ロケーションに *flow* 条件（クロック変数の傾き）を割り当てる関数 Γ: $L \to Flow(X_c)$
- アクションの有限集合 $Act = Act_{in} \cup Act_{out}$
- ロケーションに履歴指定を割り当てる関数 $Hist$: $L \to \{\varepsilon,\ H,\ H^*\}$
- 遷移規則の有限集合 $T \subseteq L \times Act_{in} \times Guard(X) \times Update(X) \times 2^L \times Act_{out} \times L$
  遷移規則の要素 $(l, a_{in}, \phi, up, cl, a_{out}, l') \in T$ は次で表される．
  - $l,\ l' \in L$ は遷移元および遷移先のロケーション．
  - $a_{in} \in Act_{in}$ は入力アクション．
  - $\phi \in Guard(X)$ はガード条件．
  - $up \in Update(X)$ は更新式のリスト．
  - $cl \subseteq L$ は履歴情報を消去するロケーションの集合．
  - $a_{out} \in Act_{out}$ は出力アクション．

□

また，履歴指定と履歴消去については，次のように定義する．

定義 2.18（履歴指定）
履歴指定関数 $Hist$: $L \to \{\varepsilon,\ H,\ H^*\}$ は，ロケーションに履歴の種類を割り当てる．ロケーション l に H が割り当てられた場合，l が非アクティブになるときの最後の l のサブロケーション $l^{sub} \in h(l)$ が履歴に記録される．また，H^* を割り当てられた場合は，基底ロケーションに至るまでのすべてのアクティブなロケーションが履歴に記録される．

□

定義 2.19（履歴消去）
履歴消去は，次の構文で定義される．

$$\begin{aligned} ClearHistory ::= {} & clear(l) \mid deep\text{-}clear(l) \\ & \mid ClearHistory_1\ ,\ ClearHistory_2 \end{aligned}$$

$clear(l)$ は $Hist(l) = H$ の場合に，$deep\text{-}clear(l)$ は $Hist(l) = H^*$ の場合に使用される．また履歴消去の意味 $[\![ClearHistory]\!]$ は消去するロケーションの集合である．それは次のように定義される．

- $clear(l)$ の場合，$l \in [\![ClearHistory]\!]$ である．
- $deep\text{-}clear(l)$ の場合，l の支配下にある子孫ロケーションの集合 $L^* = \{l' \mid l' \in h(l^*) \wedge (l^* \in L^* \vee l^* = l)\}$ について，$L^* \cup \{l\} \subseteq [\![ClearHistory]\!]$ である．

□

図 1 は，線形階層ハイブリッドオートマトンを図式で書いた例である．線形階層ハイブリッドオートマトンは角丸の四角で囲われ，内部にロケーションを複数持つ．遷移は矢印で描かれ，遷移時の条件が矢印の上に描かれる．また，丸囲みの “H” は，ロケーションに対する履歴指定である．

図 1 の例では，まず初期ロケーション L1 から開始される．アクション *start* を受け取ると，ロケーション L2 へ遷移する．ロケーション L2 は階層ロケーションであるため，階層下の初期ロケーション L21 へ自動的に遷移される．アクション *stop* を受け取った場合，ロケーション L3 へ遷移する．このとき，出発元のロケーション L2 には履歴指定があるため，ロケーション L2 の階層下で最後に実行されていたロケーション（ここではロケーション L22 とする）が記憶される．ロケーション L3 から L1 に戻り，再度ロケーション L2 に遷移した場合，記憶された階層下のロケーション L22 から処理は再開される．

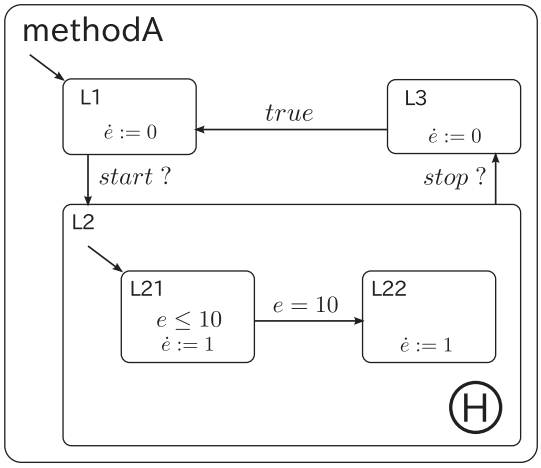

図 1　線形階層ハイブリッドオートマトンの記述例
Fig. 1　Hierarchical Linear Hybrid Automaton example.

履歴の動作と評価については 2.4.2 項に後述する．

2.3.2　クラス/オブジェクトの構文

クラスはシステムのモジュールの設計を表現し，オブジェクトは稼働中のモジュールを表現する．また，システムの再構成はオブジェクト単位で実行される．ここでは，クラスとオブジェクトの構文を定義する．

クラスには，動的クラスと静的クラスの 2 つがある．静的クラスはシステムを構成する基本モジュールを，動的クラスはシステムで動的構成されるモジュールを表す．

定義 2.20（クラスの定義）
動的クラス cls_{dyn} および静的クラス cls_{std} の構文 $\langle HLHAs, X, Init\rangle$ は，次の要素からなる．

- 線形階層ハイブリッドオートマトンの有限集合 $HLHAs$
- 変数の有限集合 $X = X_c \cup X_d \cup X_r \cup X_l$
- 変数の初期割当てを指定する関数 $Init$: $X \to \{0\}^{X_c} \cup \mathbb{Z}^{X_d} \cup (Objects_{std} \cup \{null\})^{X_r} \cup \{[\,]\}^{X_l}$

ここで，クラスに含まれる変数集合 X は，$HLHAs$ に含まれる変数集合の和集合となる．また，動的クラスの集合を $Classes_{dyn}$，静的クラスの集合を $Classes_{std}$，クラス全体の集合を $Classes$ とする．

□

図 2 は静的クラスを，また**図 3** は動的クラスを図式で描いた例である．図式で記述する場合，動的クラスの場合のみクラス名を山括弧で囲む．クラスは四角で囲まれ，右

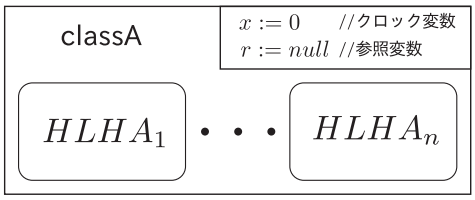

図 2　静的クラスの記述例
Fig. 2　Static Class example.

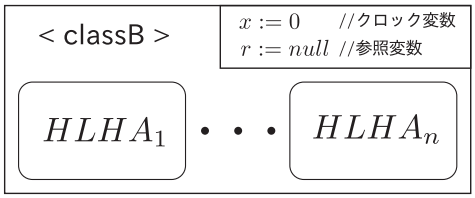

図 3　動的クラスの記述例
Fig. 3　Dynamic Class example.

上にクラスが保持する変数が，また中にはクラスが保持する線形ハイブリッドオートマトンが複数個記述される．

クラスと同じように，オブジェクトにも動的オブジェクトと静的オブジェクトの 2 つがある．静的オブジェクトはシステムで静的構成されるモジュールを，動的オブジェクトはシステムで動的構成されるモジュールを表す．

定義 2.21（オブジェクトの定義）
動的オブジェクト obj_{dyn} および静的オブジェクト obj_{std} の構文 $\langle HLHAs, X, Init\rangle$ は，次の要素からなる．

- 線形階層ハイブリッドオートマトンの有限集合 $HLHAs$
- 変数の有限集合 $X = X_c \cup X_d \cup X_r \cup X_l$
- 変数の初期割当てを指定する関数 $Init$: $X \to \{0\}^{X_c} \cup \mathbb{Z}^{X_d} \cup (Objects_{std} \cup \{null\})^{X_r} \cup \{[\,]\}^{X_l}$

また，動的オブジェクトの集合を $Objects_{dyn}$，静的オブジェクトの集合を $Objects_{std}$，オブジェクト全体の集合を $Objects$ とする．

□

動的オブジェクトは動的クラスから，静的オブジェクトは静的クラスから生成される．2.2.2 項で上述したとおり，動的クラスと動的オブジェクトは 1 対多の対応関係となる．対して，静的オブジェクトと静的クラスは 1 対 1 の対応関係となる．そのため，変数の初期化関数 $Init$ では，静的クラスを割り当てた場合，対応する静的オブジェクトを割り当てたと解釈するとする．また，オブジェクトはクラスから生成されるため，オブジェクトを図式で表す必要はない．

各オブジェクトが持つ変数とアクションの影響範囲は次のようになる．

定義 2.22（変数とアクションの影響範囲）
オブジェクトが持つ変数とアクションは，そのオブジェクトに含まれる線形階層ハイブリッドオートマトンのみに影響するとする．別のオブジェクトへ影響させる場合には，参照変数を用いて影響させるとする．

□

この定義は，変数やアクションが他のオブジェクトへ与える影響を限定するために必要である．もし他のオブジェクトと変数やアクションを共有したい場合は，参照変数を用いる必要がある．

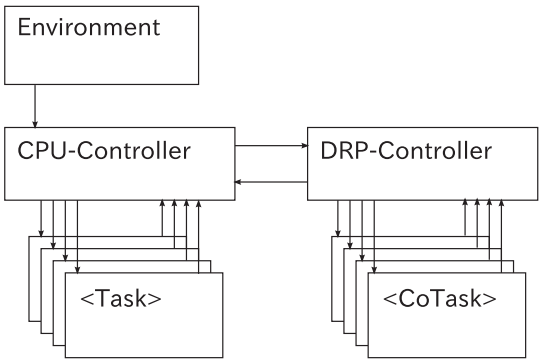

図 4　システムの記述例
Fig. 4　System example.

2.3.3　システムの構文

最後に，システム全体の構造を表現するシステムを定義する．

定義 2.23（システムの構文）
システムの構文 $System = \langle Classes, Objects\rangle$ は，次の要素からなる．

- クラスの有限集合 $Classes = Classes_{dyn} \cup Classes_{std}$
- オブジェクトの有限集合 $Objects = Objects_{dyn} \cup Objects_{std}$

□

図 4 はシステムを図式で表した例である．図 4 では，3 つの静的クラスと 8 つの動的クラスから構成されている．なお，実際のクラス間の接続関係はクラスの参照変数に対する割当てで決定される．そのため，図 4 の接続関係（矢印）は自動で導出可能である．

2.4　意味

動的再構成可能システムの動作は，コンフィギュレーションの遷移系として定義される．ここでは，まずコンフィギュレーションを定義し，それを用いて線形階層ハイブリッドオートマトンの動作を定義する．その後に，クラスとオブジェクト，システムの動作を定義する．

2.4.1　コンフィギュレーション

コンフィギュレーションとは，現在アクティブなロケーションの集合である．

定義 2.24（コンフィギュレーション）
ロケーション l のアクティブなサブロケーションを返す関数 ρ: $L \to L$ を用いて得られる，ある時点でのアクティブな最大のロケーション集合 $c = \rho^*(root)$ をコンフィギュ

レーションと呼ぶ．ただし，ρ は以下を満たす関数である．

$$\rho(l) = \begin{cases} \bot \text{ (undefined)} & (type(l) = BASIC) \\ l' & (type(l) = OR,\ l' \in h(l), \\ & \quad l' \text{ is active}) \end{cases}$$

また，ρ^*: $L \to 2^L$ は ρ を用いて次のように定義される関数である．

$$\rho^*(l) = \begin{cases} \{l\} & (type(l) = BASIC) \\ \{l\} \cup \rho^*(l') & (type(l) = OR,\ l' = \rho(l)) \end{cases}$$

さらに，コンフィギュレーションの全体集合を C，すべての要素が初期ロケーション集合に含まれるコンフィギュレーション $c_0 \subseteq L_0$ を初期コンフィギュレーションと呼ぶ．□

2.4.2　履歴と動作

線形階層ハイブリッドオートマトンではロケーションに履歴指定をすることができる．ここでは，履歴の評価と動作を定義する．

定義 2.25（履歴評価）
履歴評価 ξ: $L \to L \cup \{\varepsilon\}$ は，任意のロケーション l にロケーションまたは空要素を割り当てる関数である．また，$\xi[L' := l^*]$ によって，集合 $L' \subseteq L$ の要素の評価値を $l^* \in L \cup \{\varepsilon\}$ に，それ以外については ξ と同じ評価値を割り当てることを表す．

□

履歴評価は，サブロケーションの履歴情報を親ロケーションに割り当てる関数である．コンフィギュレーションは履歴の影響により変更される場合がある．

定義 2.26（コンフィギュレーション変更関数）
コンフィギュレーション変更関数 ζ: $C \times \xi \to C$ は，履歴評価 ξ を用いて，コンフィギュレーションに履歴評価を反映する関数である．関数 ζ は次のように定義される．

$$\zeta(c) = \zeta'(c, l_0^{root}) \cup \cdots \cup \zeta'(c, l_n^{root})$$
$$\because\ l_i^{root} \notin h(l_k)\ \wedge\ l_i^{root},\ l_k \in c$$
$$\zeta'(c,l) = \begin{cases} l \cup \zeta'(L_0, \xi(l)) & (\xi(l) \neq \varepsilon) \\ l \cup \zeta'(c, l') & (\xi(l) = \varepsilon \wedge l' \in h(l) \\ & \quad \wedge\ l' \in c) \\ l & (h(l) = \emptyset) \end{cases}$$

□

コンフィギュレーション変更関数 ζ は，コンフィギュレーション c のうち最上位に位置するロケーション l^{root} から順に，履歴評価に含まれるかを調べる．含まれない場合はロケーション l^{root} の階層下のロケーションを，含まれる場合は $\xi(l_{root})$ を対象に再帰的に調べ，変更を行う．なお，L_0 は初期ロケーション集合であり，階層下の初期ロケーションを導出するために使用している．

2.4.3　優先遷移指定

遷移 $(l, a_{in}, \phi, up, cl, a_{out}, l') \in T$ について，条件 ϕ に *asap* が含まれる場合，その遷移は優先的に実行されることを示す．2.4.4 項で定義される 3 種類の遷移（時間遷移 $\Rightarrow_\delta$ と離散遷移 $\Rightarrow_d$ および連続遷移 $\Rightarrow_{d\star}$）について，*asap* を含む離散遷移が成功した場合，時間遷移は行われない．また，*asap* は連続遷移では考慮されない．

2.4.4　線形階層ハイブリッドオートマトンの意味

線形階層ハイブリッドオートマトンの動作は，コンフィギュレーション間を遷移する遷移系ととらえることができる．遷移系を定義する準備として，コンフィギュレーションを用いて階層のないオートマトンに変換（平坦化）した線形階層ハイブリッドオートマトンの構文を定義する．

定義 2.27（平坦化した線形階層ハイブリッドオートマトン）　平坦化した線形階層ハイブリッドオートマトン $HLHA^f = \langle C, c_0, X, I^f, \Gamma^f, Act^f, T^f \rangle$ は，次の要素からなる．

- コンフィギュレーションの集合 C
- 初期コンフィギュレーション $c_0 \in C$
- 変数の有限集合 $X = X_c \cup X_d \cup X_r \cup X_l$
- コンフィギュレーションに不変条件を割り当てる関数 I^f: $C \to Guard(X)$
- コンフィギュレーションに *flow* 条件（クロック変数の傾き）を割り当てる関数 Γ^f: $C \to Flow(X_c)$
- アクションの冪集合 $Act^f = 2^{Act_{in}} \cup 2^{Act_{out}}$
- 遷移規則の有限集合 $T^f \subseteq C \times 2^{Act_{in}} \times Guard(X) \times Update(X) \times 2^L \times 2^{Act_{out}} \times C$
  その要素は $(c, a_{in}^f, \phi, up, cl, a_{out}^f, c') \in T_f$ で表される．

□

アクションが冪集合になったのは，次に定義するオートマトンの並列合成で，遷移規則に複数のアクションを持つ必要があるからである．なお，$a_{in}^f \in 2^{Act_{in}}$ また $a_{out}^f \in 2^{Act_{out}}$ であり，集合に含まれるアクションはすべて同時に満たされなければならない．

複数の線形階層ハイブリッドオートマトンの並列合成について定義する．

定義 2.28（線形階層ハイブリッドオートマトンの並列合成）　2 つの線形階層ハイブリッドオートマトン $HLHA_1$，$HLHA_2$ の並列合成は，平坦化した線形階層ハイブリッドオートマトンの並列合成 $HLHA_1^f \parallel HLHA_2^f = \langle C^{\parallel}, c_0^{\parallel}, X^{\parallel}, I^{\parallel}, \Gamma^{\parallel}, Act^{\parallel}, T^{\parallel} \rangle$ によって定義される．

- $C^{\parallel} = \bigcup_{c_1 \in C_1} \bigcup_{c_2 \in C_2} \{c_1 \cup c_2\}$
- $c_0^{\parallel} = c_{0_1} \cup c_{0_2}$
- $X^{\parallel} = X_1 \cup X_2$
- $I^{\parallel} : c_1 \cup c_2 \mapsto I_1^f(c_1) \wedge I_2^f(c_2)$
- $\Gamma^{\parallel}$ は以下を満たす．

$$\Gamma^{\parallel}(c_1 \cup c_2)(x) =$$

$$\begin{cases} \Gamma_1^f(c_1)(x) & (x \in X_{c1} \wedge x \notin X_{c2}) \\ \Gamma_2^f(c_2)(x) & (x \notin X_{c1} \wedge x \in X_{c2}) \\ \Gamma_1^f(c_1)(x) & (x \in X_{c1} \wedge x \in X_{c2} \\ & \quad \wedge \Gamma_1^f(c_1)(x) = \Gamma_2^f(c_2)(x)) \\ \bot \text{ (undefined)} & \text{(otherwise)} \end{cases}$$

- $Act^{||} = Act^{||}_{in} \cup Act^{||}_{out}$
  ここで，$Act^{||}_{in}$ と $Act^{||}_{out}$ は以下を満たす．

$$Act^{||}_{in} = 2^{Act_{in_1}} \cup 2^{Act_{in_2}}$$
$$Act^{||}_{out} = 2^{Act_{out_1}} \cup 2^{Act_{out_2}}$$

- $T^{||} \subseteq C^{||} \times Act^{||}_{in} \times Guard^{||}(X^{||}) \times Update^{||} \times 2^{L_1 \cup L_2} \times Act^{||}_{out} \times C^{||}$
  ここで，$Guard^{||}(X^{||})$ と $Update^{||}(X^{||})$ は以下を満たす．
  - $Guard^{||}(X^{||}) = \{\phi_1 \wedge \phi_2 \mid \phi_i \in Guard_i(X_i)\}$.
  - $[u_0^1, u_1^1, \cdots, u_m^1] \in Update_1(X_1)$，$[u_0^2, u_1^2, \cdots, u_n^2] \in Update_2(X_2)$ について，$Update^{||}(X^{||})$ はリストの連結で定義される．

    $$[u_0^1, u_1^1, \cdots, u_m^1] :: [u_0^2, u_1^2, \cdots, u_n^2] \in Update^{||}(X^{||})$$

    ただし，更新式 u_k^1（$0 \leq k \leq m$）と u_l^2（$0 \leq l \leq n$）の間に順序依存性があってはならない．
    また，遷移規則は次の手順で合成される．ここで，$(c_i, a^f_{in_i}, \phi_i, up_i, cl_i, a^f_{out_i}, c'_i) \in T_i^f$ とする．
  - $a^f_{in_1} \cap a^f_{in_2} \neq \emptyset$ のとき，

    $$(c_1 \cup c_2, a^f_{in_1} \cup a^f_{in_2}, \phi_1 \wedge \phi_2, up_1 :: up_2, cl_1 \cup cl_2, a_{out_1} \cup a_{out_2}, c'_1 \cup c'_2) \in T^{||}$$

  - $a^f_{in_1} \cap a^f_{in_2} = \emptyset$ のとき，

    $$(c_1 \cup c_2, a^f_{in_1}, \phi_1, up_1, cl_1, a^f_{out_1}, c'_1 \cup c_2) \in T^{||}$$
    $$(c_1 \cup c_2, a^f_{in_2}, \phi_2, up_2, cl_2, a^f_{out_2}, c_1 \cup c'_2) \in T^{||}$$

□

コンフィギュレーションと平坦化を用いることで，線形階層ハイブリッドオートマトンの状態と動作を考えることができる．

定義 2.29（線形階層ハイブリッドオートマトンの状態）
線形階層ハイブリッドオートマトンの状態 $q = \langle c, \nu, \xi \rangle$ は次の要素からなる．

- コンフィギュレーション $c \in C$
- 変数評価 ν: $X \to \nu_c \cup \nu_d \cup \nu_r \cup \nu_l$
  なお，変数評価 ν は各変数集合についての評価に分割できる．
  - クロック評価 ν_c: $X_c \to \mathbb{R}_{\geq 0}$
  - 離散変数評価 ν_d: $X_d \to \mathbb{Z}$
  - 参照変数評価 ν_r: $X_r \to Objects \cup \{null\}$
  - リスト変数評価 ν_l: $x_l \mapsto [a_0, a_1, \cdots]$
- 履歴評価 ξ: $L \to L \cup \{\varepsilon\}$

□

定義 2.30（線形階層ハイブリッドオートマトンの意味）
線形階層ハイブリッドオートマトンの意味は，遷移系 $\mathcal{M} = \langle Q, \Rightarrow, q_0 \rangle$ として定義される．

- 状態集合 Q
- 初期状態 $q_0 \in Q$
  初期状態 q_0 は (c_0, ν_0, ξ_0) で表される状態である．オブジェクトが持つ初期化関数 $Init$ を用いると，初期変数評価値は $\nu_0 = \nu[X := Init(X)]$ で表現される．また，初期履歴評価値は $\xi_0 = \xi[L := \varepsilon]$ である．
- 遷移 $\Rightarrow$ は時間遷移 $\Rightarrow_\delta$ と離散遷移 $\Rightarrow_d$ および連続遷移 $\Rightarrow_{d\star}$ の和集合
  - 時間遷移 $\Rightarrow_\delta$
    任意の状態 $(c, \nu, \xi) \in Q$ と時間経過 $\delta \in \mathbb{R}_{\geq 0}$ に対して，$c' = c$ かつ $\nu'(x_c) = \nu(x_c) + \Gamma^f(c)(x_c) \cdot \delta$，かつ ν' が $I^f(c)$ を満たすときに限り $(c, \nu, \xi) \Rightarrow_\delta (c, \nu', \xi)$ である．
    ただし，コンフィギュレーション c からの遷移規則 $(c, a^f_{in}, \phi, up, cl, a^f_{out}, c'') \in T^f$ について，ϕ に $asap$ を含む遷移 t が存在した場合，遷移 t による離散遷移を先に行う．遷移 t の離散遷移が成功した場合，時間遷移は行われない．
  - 離散遷移 $\Rightarrow_d$
    任意の状態 $(c, \nu, \xi) \in Q$ に対し，$(c, \emptyset, \phi, up, cl, a^f_{out}, c') \in T^f$ が存在し，ν が ϕ を満たす場合に限り，次の更新操作を行う．
    * 更新式 $[u_0, u_1, \cdots] = up$ を順次実行し，評価値 ν を更新する．すべて実行後の評価値を ν' とする．
    * 遷移前のロケーション $\{l_0, l_1, \cdots, l_m\} = c - c'$ に含まれるロケーション l_k について，
      (1) l_k の親ロケーション l' について $Hist(l') = H$ である場合[*1]
      (2) l_k の先祖ロケーション集合 L^* の中に $Hist(l^*) = H^*$ $(l^* \in L^*)$ となるロケーションが存在する場合[*2]
      (1) または (2) の条件を満たすとき，l_k の親ロケーション l' の評価値を $\xi[\{l'\} := l_k]$ に更新する．すべての履歴を更新した後，履歴消去 $\xi[cl := \varepsilon]$ を行う．すべて更新した後の評価値を ξ' とする．
    更新操作を実行後，コンフィギュレーションに履歴評価を反映する．反映後のコンフィギュレーションを $c'_\zeta = \zeta(c', \xi)$ とする．ν' が $I^f(c'_\zeta)$ を満たすときに限り，次の出力同期を行う．

[*1] ロケーション l_k の親ロケーションとは，$l_k \in h(l')$ となるロケーション l' のことである．
[*2] ロケーション l_k の先祖ロケーション集合とは，階層関数 h を有限回使用して l_k を導出できるロケーションの集合である．

＊ $a^f_{out} = \emptyset$ の場合，同期操作を行わず，出力同期は成功する．

＊ $a^f_{out} \neq \emptyset$ の場合，a^f_{out} に対応する a'^f_{in} を持つ遷移規則 $(c'_\varsigma, a'^f_{in}, \phi', up', cl', a'^f_{out}, c'') \in T^f$ へ強制的に連続遷移を発生させる．ただし，対応する遷移規則が複数存在した場合，非決定に遷移規則が選ばれる．連続遷移が失敗した場合，出力同期は失敗する．

出力同期が成功したときに限り，$(c, \nu, \xi) \Rightarrow_d (c'_\varsigma, \nu', \xi')$ である．

– 連続遷移 $\Rightarrow_{d\star}$

任意の状態 $(c, \nu, \xi) \in Q$ に対し，出力アクション a^{*p}_{out} の連続遷移として $(c, a^f_{in}, \phi, up, cl, a^f_{out}, c') \in T^f$ が呼び出された場合，次の同期操作を行う．

＊ $\{\alpha_i\} = a^{*p}_{out}$，$\{\beta_i\} = a^f_{in}$，また $[\![\alpha_k]\!] = (act^{out}_k, val_k)$，$[\![\beta_l]\!] = (act^{in}_l, x_l)$ について，$act^{in}_l = act^{out}_k$ である場合，更新式のリスト up の最初に更新式 $x_l := val_k$ を追加する．すべての更新式を追加したあとの更新式リストを up' とする．ただし，追加した更新式に順序依存性があってはならない．

同期操作後，$(c, \emptyset, \phi, up', cl, a^f_{out}, c')$ として離散遷移と同じ遷移判定を行う．遷移判定が成功したときに限り，$(c, \nu, \xi) \Rightarrow_{d\star} (c', \nu', \xi')$ である．

□

2.4.5　クラス/オブジェクトの意味

クラスとオブジェクトの意味は，それが持つ線形階層ハイブリッドオートマトンの並列合成で定義される．

定義 2.31（クラス/オブジェクトの意味）

クラスおよびオブジェクトの意味は，それが持つ線形階層ハイブリッドオートマトンの集合 $HLHAs$ について，各オートマトンを平坦化し，すべてを並列合成したオートマトン $HLHA^f_{||}$ の意味である遷移系 $\mathcal{M}_{||}$ である．

□

2.4.6　システムの意味

システムの意味は，それが持つオブジェクトの意味の合成で定義される．

定義 2.32（システムの意味）

システムの意味は，システムが持つオブジェクト集合 $Objects$ の各遷移系 $\mathcal{M}_{||}$ の合成である遷移系 $\mathcal{M}_{sys}$ で表される．なお，この合成は，各オブジェクトが持つ参照変数を通して変数およびアクションがオブジェクト間で共有された場合の線形階層ハイブリッドオートマトンの並列合成として定義する．

□

2.5　到達可能性問題

本稿では，検証項目としてシステムの重要な性質である安全性を扱う．安全性は到達可能性解析によって検証することができる．ここでは，到達可能性問題について定義する．

定義 2.33（パス）

遷移系 $\mathcal{M} = (Q, \Rightarrow, q_0)$ 上の初期状態からの有限または無限列 $\omega = q_0 \Rightarrow q_1 \Rightarrow q_2 \Rightarrow \cdots$ をパスと定義する．

□

定義 2.34（到達可能性問題）

システム $System = (Classes, Objects)$ の到達可能性問題とは，ターゲットロケーションの集合 L_{target} に到達するパス ω が存在するかどうかである．$System$ の遷移系 $\mathcal{M}_{sys} = (Q, \Rightarrow, q_0)$ 上でターゲットロケーション $l_{target} \in L_{target}$ を含むコンフィギュレーション c_{target}（ターゲットコンフィギュレーション）を持つ状態 $q_{target} = (c_{target}, \nu, \xi)$ に到達するパス ω が存在する場合，かつその場合に限り，出力は “$reachable$” となり，それ以外は “$not\ reachable$” となる．

□

3.　仕様記述の事例

この章では，本稿で提案した仕様記述言語を用いて，動的再構成可能システムのモデルを記述する．また，次章では作成したモデルに対して到達可能性解析を行う．

3.1　モデルの方針

動的再構成可能システムとして，本稿では CPU と DRP を組み合わせたシステムを取り扱う．DRP は粗粒度再構成可能アーキテクチャであり，一般的な FPGA と比べて再構成が速く，また実行時に再構成が可能という特徴を持つ．DRP の再構成はタイル（論理セル）単位で実行される．

本稿では，CPU 側でソフトウェアタスクを 1 つ取り扱い，DRP 側でハードウェアタスクを複数個取り扱うシステムを考える．また，各装置における制約を次のようにする．

CPU の制約

- 各 CPU タスクが持つ優先度を用いたプリエンプションありの優先度スケジューリングを採用する．
- CPU タスクは同時に 1 つのみ実行される．また，周波数は固定である．
- CPU タスクが DRP タスクを生成した場合，明示的に I/O 待ち状態にならない限り，実行は継続される．

DRP の制約

- 到着順スケジューリングとする．
- 各 DRP タスクは占有タイル数と実行可能最大周波数を持つ．
- タイルが足りている場合に限り DRP タスクを複数個実行する．また，実行周波数は実行中の DRP タスクの実行可能最大周波数の最小値に変更される．実行中の DRP タスクがない場合の実行周波数は 0 とする．

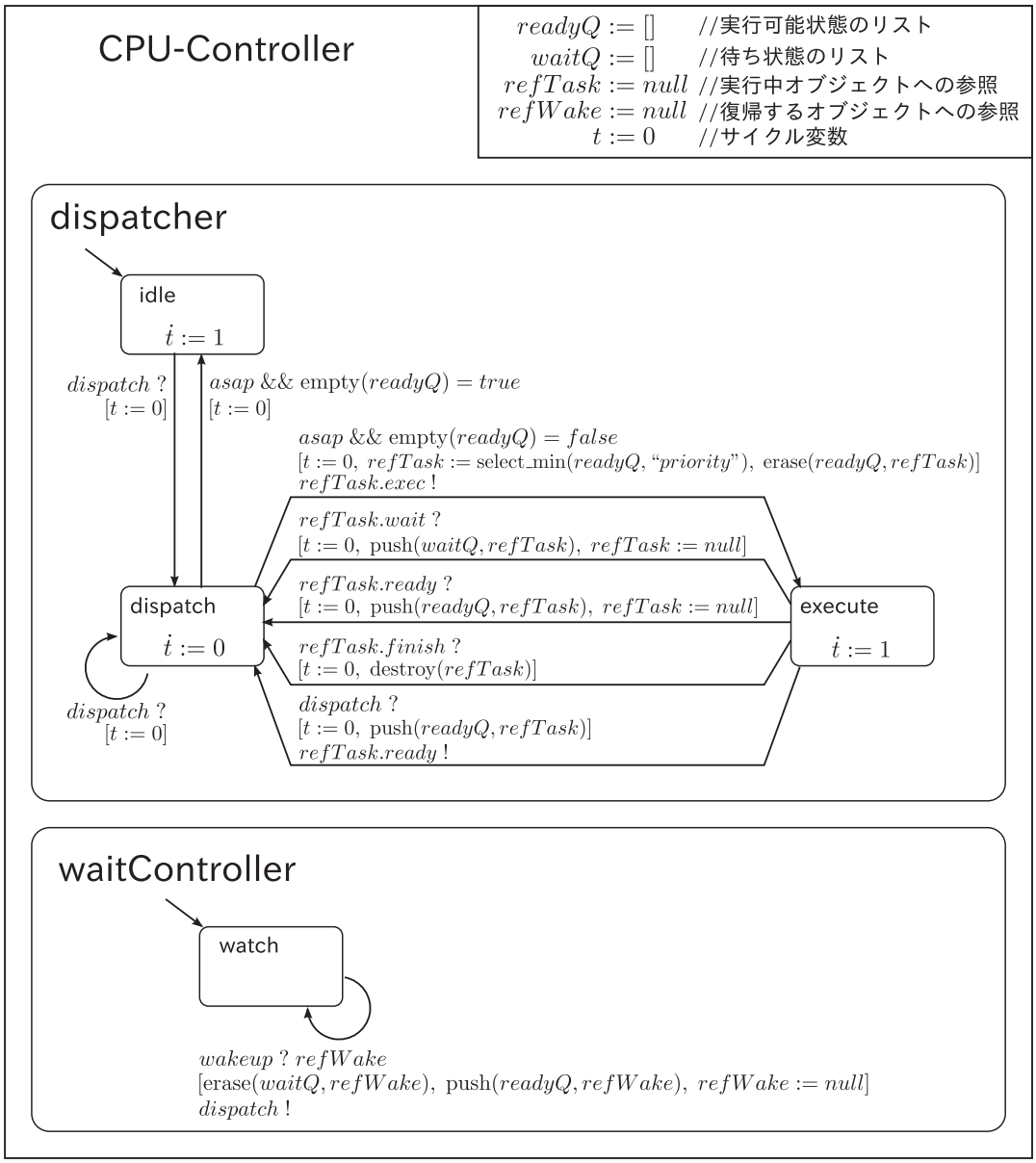

図 5　静的クラス CPU-Controller
Fig. 5　Static Class: CPU-Controller.

- DRP の再構成（DRP タスクの生成・破棄）は瞬時に行われるとする．

3.2　事前準備

本稿で記述するモデルでは，リストに対して新しい操作関数が必要になる．追加の操作関数を次のように定義する．

- （要素削除）erase: $X_l \times X$
  リスト変数 $x_l = [a_0, a_1, \cdots, a_n]$ と変数 y を受け取り，$a_k = y$ となる要素を除き，x_l の評価値を $[a_0, a_1, \cdots, a_{k-1}, a_{k+1}, \cdots, a_n]$ に更新する．この関数は更新式で使用される．
- （最小値取得）value_min: $X_l \times \mathit{Strings} \to \mathbb{Z}_{\geq 0}$
  オブジェクトを保持するリスト変数 $x_l = [a_0^{obj}, a_1^{obj}, \cdots, a_n^{obj}]$ と文字列 str を受け取り，評価値が最小となる変数 $a_k^{obj}.str$ を選択し，その変数の評価値を返す．ただし，最小となる評価値が 0 より小さい場合は 0 を返す．この関数は *flow* 条件で使用される．
- （最小値要素選択）select_min: $X_l \times \mathit{Strings} \to \mathit{Objects}$
  オブジェクトを保持するリスト変数 $x_l = [a_0^{obj}, a_1^{obj}, \cdots, a_n^{obj}]$ と文字列 str を受け取り，変数 $a_k^{obj}.str$ の評価値が最小となるオブジェクト $a_{k'}^{obj}$ を返す．この関数は更新式で使用される．

3.3　全体構成

今回記述するモデルは，図 4 に示すように，3 つの静的クラスと 4 つの動的クラスから構成される．

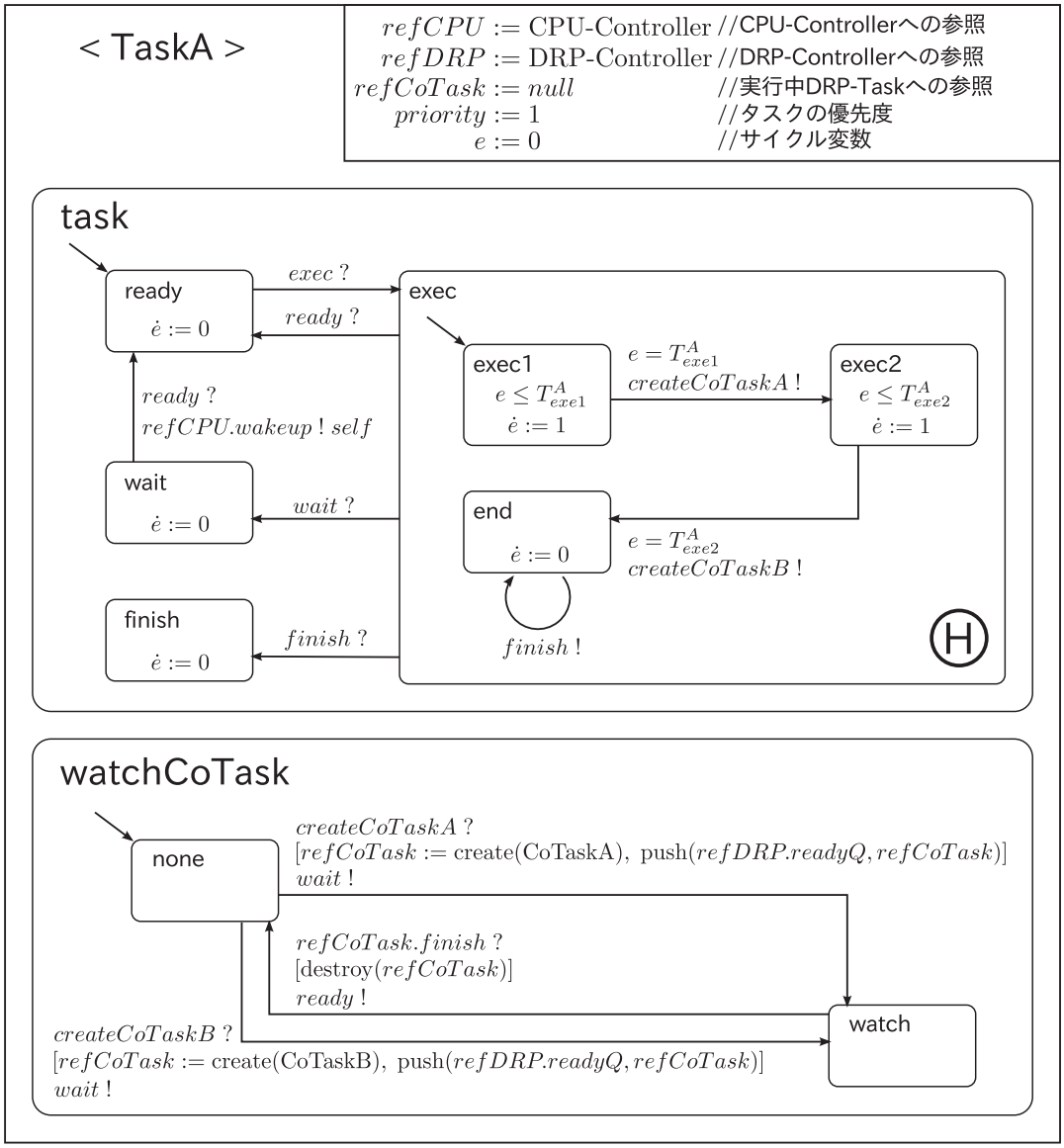

図 6　動的クラス TaskA
Fig. 6　Dynamic Class: TaskA.

- CPU-Controller
  CPU の挙動を表す静的クラス．CPU タスクのスケジューリングや各 CPU タスクの待ち状態監視を行う．
- TaskA，TaskB
  CPU タスクの挙動を表す動的クラス．各 CPU タスクは後述の DRP タスクと協調動作を行う．
- DRP-Controller
  DRP の挙動を表す静的クラス．DRP タスクのスケジューリングや動作周波数の管理などを行う．
- CoTaskA，CoTaskB
  DRP タスクの挙動を表す動的クラス．CPU タスクにより生成され，CPU タスクと協調動作を行う．
- Environment
  CPU タスクの起動パターンを表す静的クラス．CPU タスクを周期的に生成する．

3.4　CPU-Controller

CPU-Controller を図 5 に示す．静的クラス CPU-Controller は 2 つのオートマトン dispatcher と waitController から構成される．

オートマトン dispatcher では CPU タスクのスケジューリングと実行の管理を行う．実行可能状態リスト *readyQ* へ CPU タスクが追加された場合またはディスパッチ要求（*dispatch*）を取得した場合，瞬時にディスパッチを行い，*readyQ* の中で “priority” が最小の CPU タスクを取得して実行状態へ遷移する．実行中の CPU タスクから実行可能要求（*ready*）・待ち要求（*wait*）・終了要求（*finish*）やディスパッチ要求を受け取った場合，それぞれのメッセージに応じてオブジェクトの保存や破棄を行い，再ディスパッチを実行する．また，ディスパッチ中に *readyQ* が空であった場合，ディスパッチが実行できないため，CPU はアイド

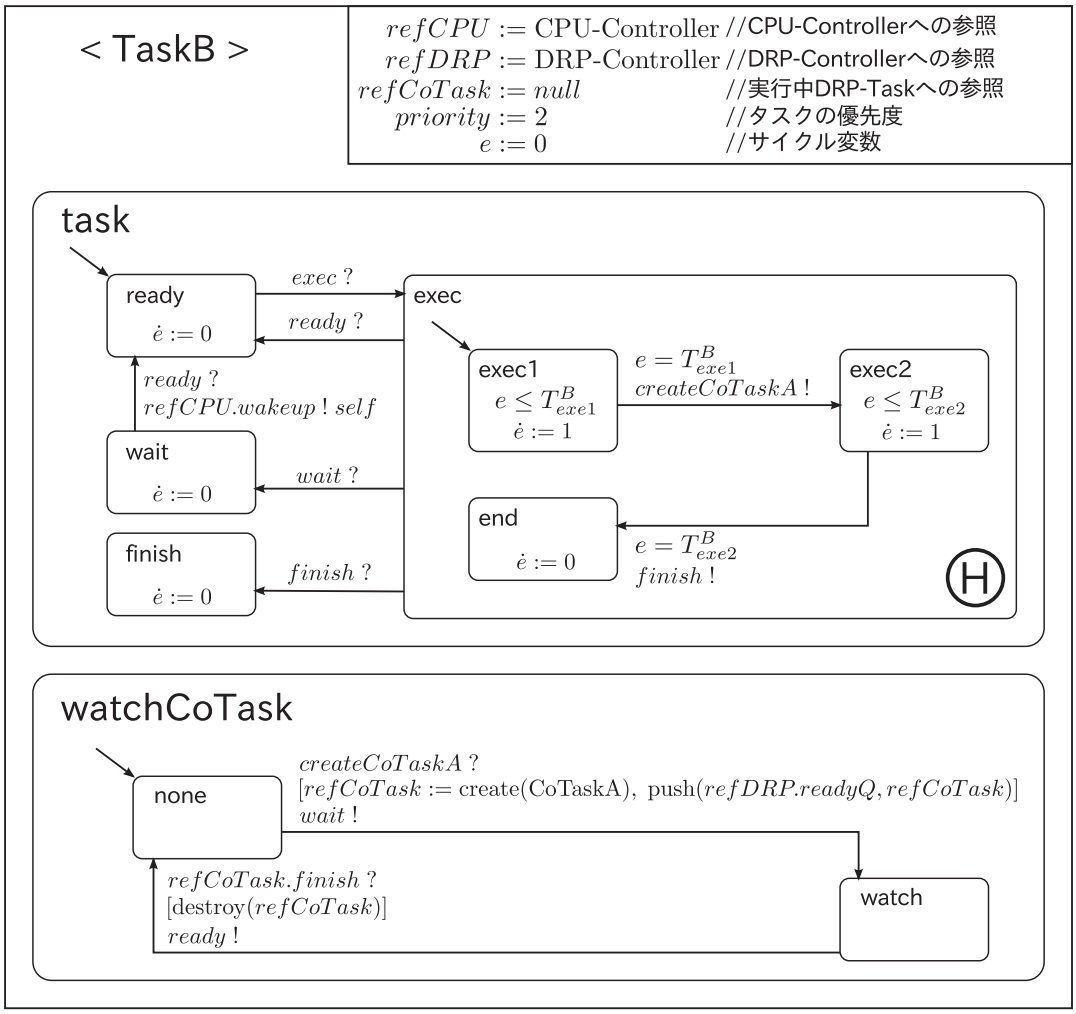

図 7　動的クラス TaskB
Fig. 7　Dynamic Class: TaskB.

ル状態に遷移する．

オートマトン waitController は待ち状態の CPU タスクからの復帰要求を受け付ける．待ち状態からの復帰要求（*wakeup*）を取得した場合，対象の CPU タスクを待ち状態リスト *waitQ* から除外し，実行可能状態リスト *readyQ* へ追加する．その後，ディスパッチ要求（*dispatch*）を発行して，監視状態に戻る．なお，ディスパッチ要求は上記したオートマトン dispatcher が受け取り，CPU タスクの再ディスパッチが実行される．

3.5　TaskA，TaskB

TaskA を**図 6** に，TaskB を**図 7** にそれぞれ示す．動的クラス TaskA および TaskB は 2 つのオートマトン task と watchCoTask から構成される．

オートマトン task は CPU タスクの処理を表す．CPU-Controller から実行要求（*exec*）を受け取ると，ロケーション exec1 に遷移をする．そして，次のロケーション exec2 に遷移するときに，DRP タスクの生成要求（*createCoTaskA*）を発行する．このメッセージは，後述のオートマトン watchCoTask が受け取り，DRP タスクの生成が行われる．exec2 に遷移後，すぐにオートマトン watchCoTask が待ち要求（*wait*）を発行するため，exec2 の処理が行われる前に CPU タスクは待ち状態 wait に遷移する．その後，再び CPU-Controller から実行要求を受け取った場合，ロケーション exec には履歴指定があるため，ロケーション exec2 から処理が再開されることになる．CPU タスクの処理が終了したら，終了要求（*finish*）を発行してロケーション finish へ遷移する．なお，CPU タスクは終了要求を受け取った CPU-Controller によって破棄されることになる．

オートマトン watchCoTask は，DRP タスクの生成・破棄および監視を行う．上記のオートマトン task から生成要求（*createCoTaskA*，*createCoTaskB*）を受け取った場合，対応する DRP タスクを生成し，生成したタスクを DRP-Controller の実行可能状態リスト *readyQ* に追加する．その後，自身の CPU タスクを待ち状態にし，ロケーション watch に遷移して DRP タスクの監視を行う．DRP タスクから終了要求（*finish*）を受け取った場合，DRP タスクを破棄し，CPU-Controller に待ち状態からの復帰要

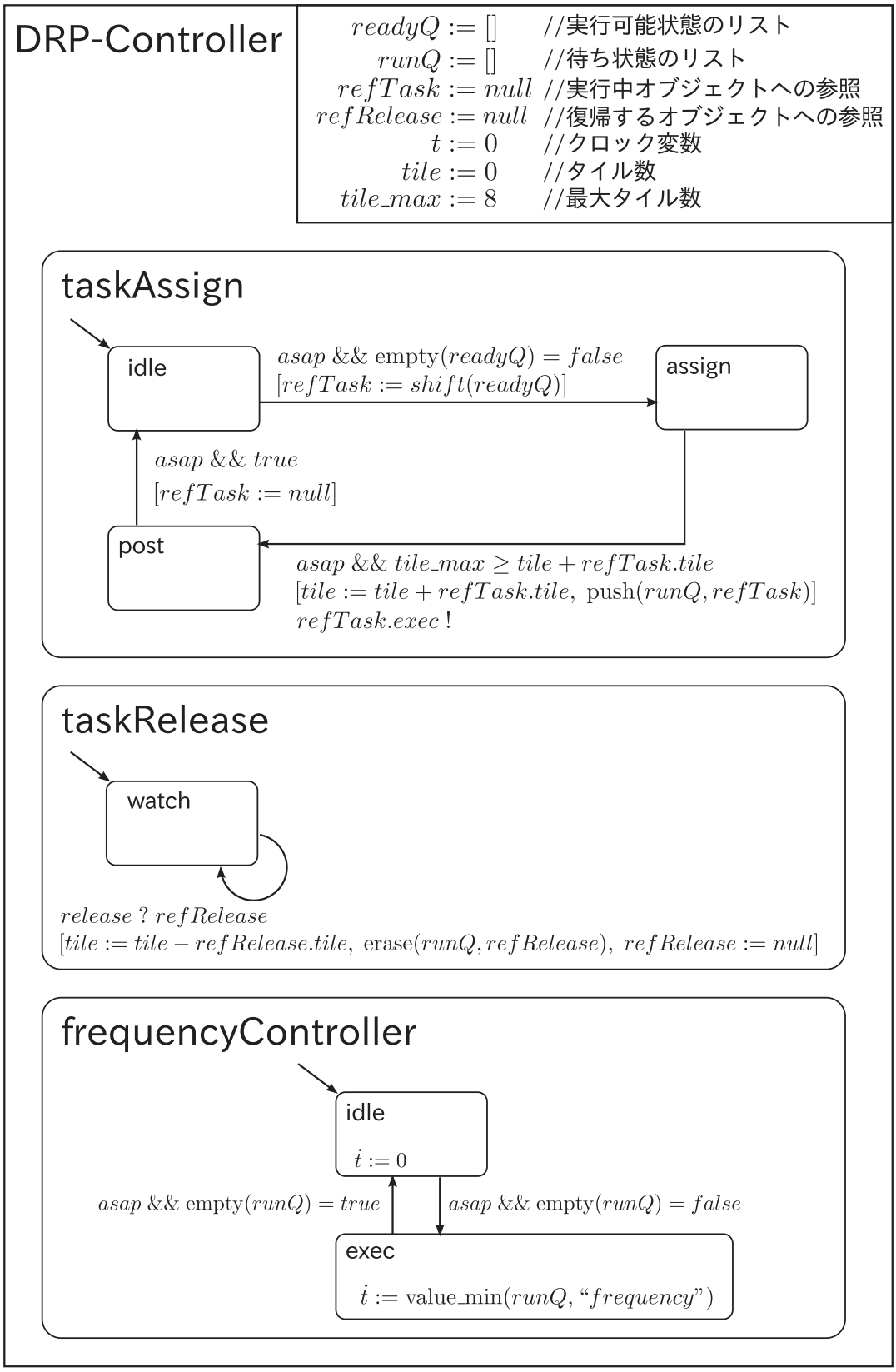

図 8　静的クラス DRP-Controller
Fig. 8　Static Class: DRP-Controller.

求（*wakeup*）を発行する．その後，ロケーション none に移動し，DRP タスクの生成メッセージを再び待つ．

なお，TaskA は DRP タスクとして *CoTaskA*，*CoTaskB* の生成要求を発生させるが，TaskB は *CoTaskA* の生成要求しか発生させない仕様となっている．

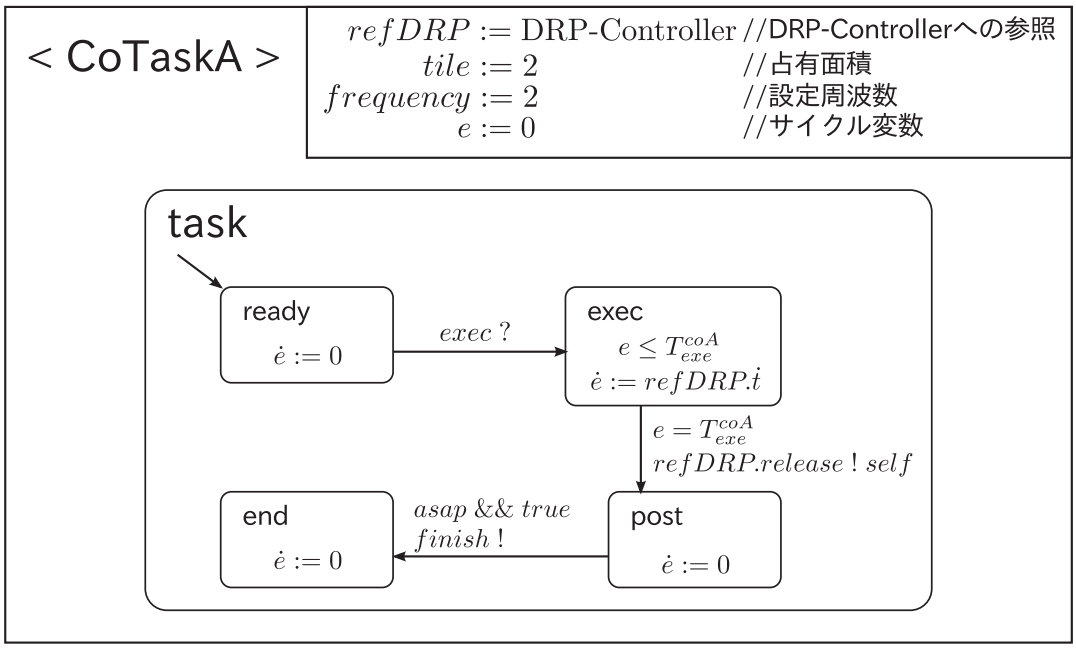

図 9　動的クラス CoTaskA
Fig. 9　Dynamic Class: CoTaskA.

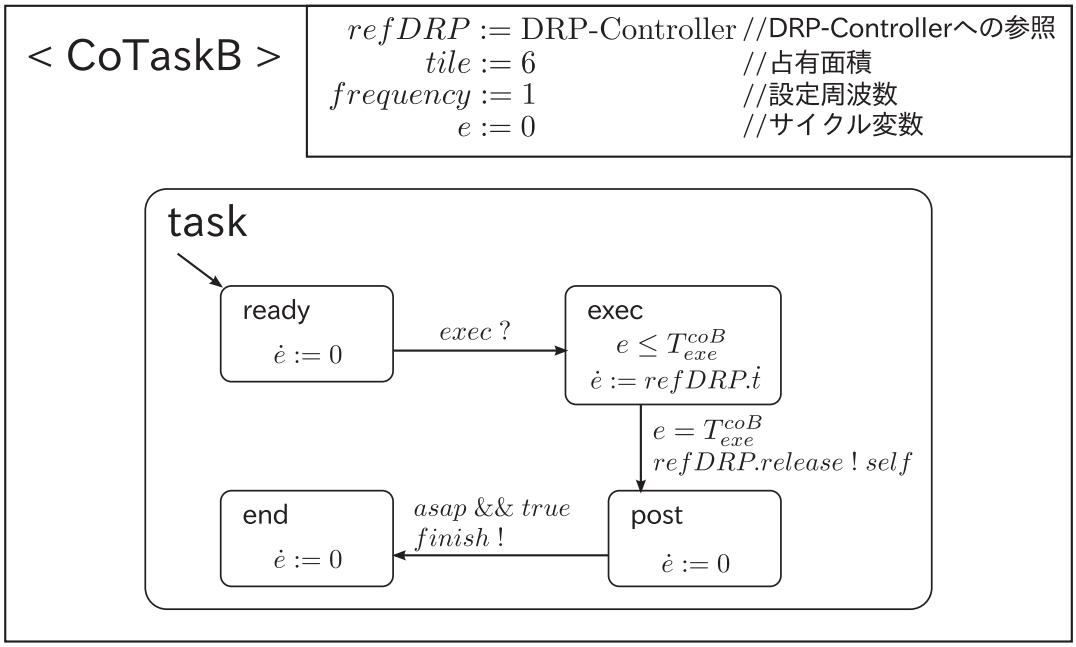

図 10　動的クラス CoTaskB
Fig. 10　Dynamic Class: CoTaskB.

3.6　DRP-Controller

DRP-Controller を図 8 に示す．静的クラス DRP-Controller は 3 つのオートマトン taskAssign，taskRelease と frequencyController から構成される．

オートマトン taskAssign は，DRP タスクの実行処理を行う．実行可能状態リスト *readyQ* に DRP タスクが追加された場合，*readyQ* の先頭から DRP タスクを取得する．そして，その DRP タスクの専有タイル数が割当て可能ならば，DRP タスクを実行中リスト *runQ* に追加し，DRP タスクに実行要求（*exec*）を発行する．

オートマトン taskRelease は，DRP タスクの終了監視を行う．DRP タスクから解放要求（*release*）を受け取った場合，対象の DRP タスクの専有タイルを解放し，実行中リスト *runQ* から DRP タスクを除外する．

オートマトン frequencyController は，DRP 全体の動作周波数を決定する．実行中リスト *runQ* が空の場合，DRP の動作周波数を 0 とする．*runQ* が空でない場合，*runQ* に含まれる各 DRP タスクが持つ “frequency” について，最

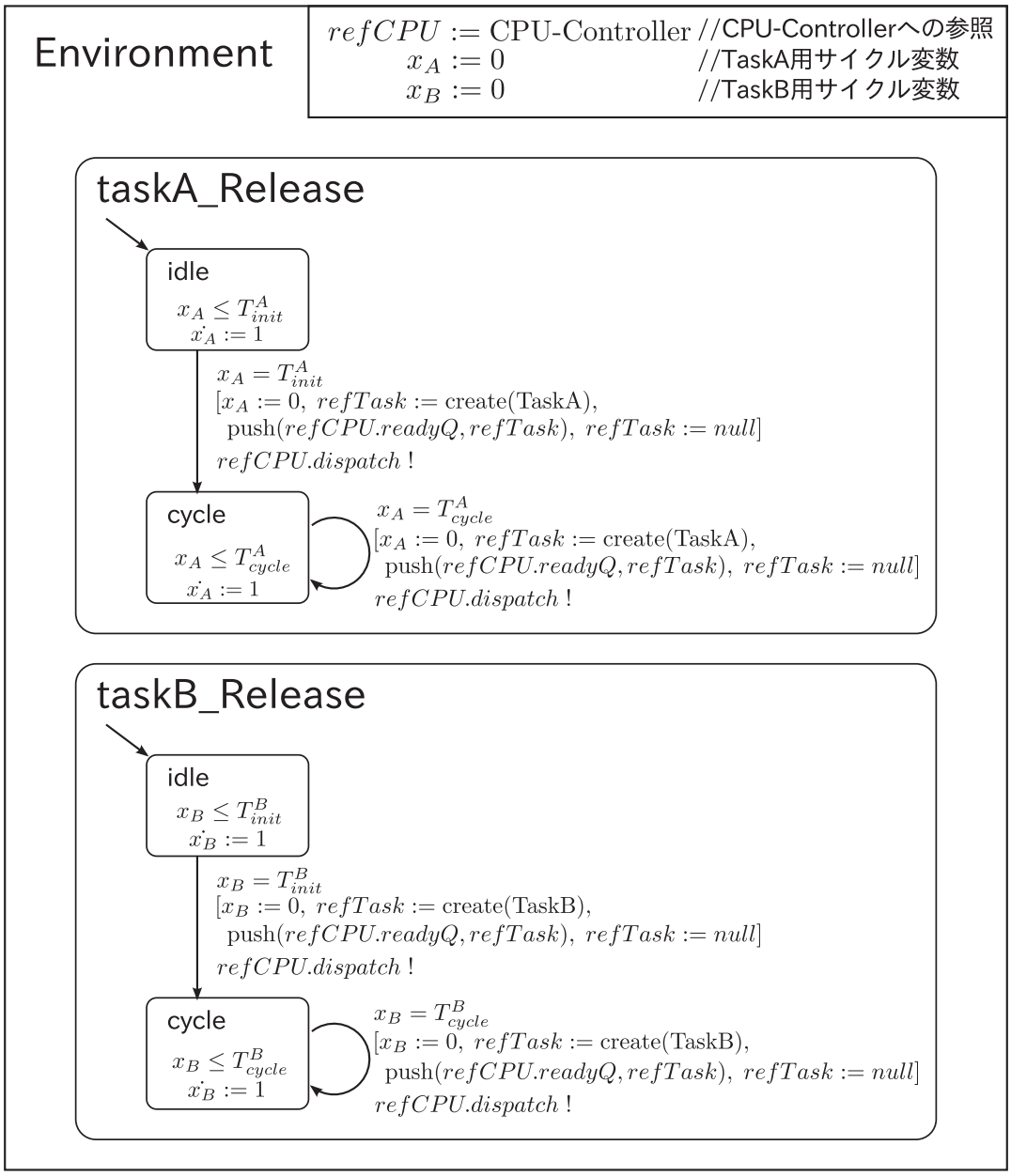

図 11　静的クラス Environment
Fig. 11　Static Class: Environment.

小の値を動作周波数とする．

3.7　CoTaskA，CoTaskB

CoTaskA を**図 9** に，CoTaskB を**図 10** にそれぞれ示す．動的クラス CoTaskA および CoTaskB はオートマトン task の 1 つで構成される．

オートマトン task は DRP タスクの処理を表す．DRP-Controller から実行要求（*exec*）を受け取ると，ロケーション exec に遷移する．ロケーション exec での処理終了後，DRP-Controller に解放要求（*release*）を発行する．その後，すぐに終了要求（*finish*）を発行し，ロケーション end へ遷移をする．なお，DRP タスクは終了要求を受け取った CPU タスクによって破棄されることになる．

CoTaskB は CoTaskA よりも専有面積が大きく，設定周波数も低い仕様となっている．

3.8　Environment

Environment を**図 11** に示す．静的クラス Environment は 2 つのクラス taskA_Release と taskB_Release から構成される．

オートマトン taskA_Release では，CPU タスク TaskA を周期的に発生させる．初期ロケーション idle で，T^A_{init} が経過した後，TaskA を生成し，生成したタスクを CPU-Controller の実行可能状態リスト *readyQ* に追加する．そして，ロケーション cycle に遷移する．その後，T^A_{cycle} が経過するごとに TaskA を生成し，CPU-Controller の *readyQ*

に追加する．また，オートマトン taskB_Release も，同じ挙動で TaskB を周期的に生成する．

4. 検証実験

この章では，提案した仕様記述言語に対する検証器を開発し，3 章で記述した動的再構成可能システムのモデルに対する検証実験（スケジューラビリティ検証）を行う．

4.1 検証実験の方針

今回のスケジューラビリティ検証は，瀧内らが提案した有限モデル検査 [15] の考えに基づいて行う．有限モデル検査とは，周期タスクのみで構成されるシステムにおいて，タスクの最大周期と，すべてのタスクの周期との最小公倍数をとり，その総和となる時間までの検査を行うことである．このようにすると，周期タスクのすべての起動タイミングの組合せが内包され，有限時間までの検証でシステムの無限の挙動を検証をしたことと等価にできる．3 章で記述したモデルでは，CPU タスクの起動タイミング T^A_{init}, T^B_{init} および起動周期 T^A_{cycle}, T^B_{cycle} を用いると，検証上限 T^{verify}_{max} は次のように計算される．ここで，LCM は最小公倍数である．

$$T^{max}_{cycle} = \max(T^A_{cycle},\ T^B_{cycle})$$
$$T^{verify}_{max} = \mathrm{LCM}(T^{max}_{cycle},\ T^A_{cycle}) + \mathrm{LCM}(T^{max}_{cycle},\ T^B_{cycle}) + \max(T^A_{init},\ T^B_{init})$$

4.2 検証アルゴリズム

今回の検証実験で実装した検証器の検証アルゴリズムの概要を Algorithm 1 に示す．検証は未到達の状態を保持する Q_{wait} と，到達済みの状態を保持する Q_{reach} を使用し，幅優先探索で到達可能状態を調べる [5]．

まず，Q_{wait} から状態 q を 1 つ取得する．それがすでに探索済みであった場合は，状態の再取得を行う．また，状態 q がターゲットロケーションを含む状態であった場合は，到達可能であるため，“*reachable*” を出力して終了する．そうでない場合は，状態 q を Q_{reach} に追加して到達済みとする．次に，状態 q から *asap* を含む離散遷移で遷移可能な遷移 $\Rightarrow_d$ があるかを調べる．遷移可能な遷移が存在するならば，それらの離散遷移後の状態を計算し，Q_{wait} に追加する．対して，*asap* を含む離散遷移が遷移不可能である場合，状態 q の時間遷移 $\Rightarrow_\delta$ を実行し状態集合 Q_δ を得る．得られた状態集合の各要素 $q' \in Q_\delta$ について，状態 q' を Q_{reach} に追加して到達済みとする．そして，状態 q' からの離散遷移（または連続離散遷移）で得られる遷移後の状態を計算し，Q_{wait} に追加する．これら操作を，Q_{wait} が空になるまで繰り返す．

すべての到達可能状態を探索してもターゲットロケーションに到達しなかった場合は，到達不可能であるため，“*not reachable*” を出力して終了する．

Algorithm 1 検証アルゴリズム（Verification algorithm）

```
1: Q_wait ← {q_0}
2: Q_reach ← ∅
3: while Q_wait is not empty do
4:    q ← shift(Q_wait)
5:    if q ∈ Q_reach then
6:       continue
7:    end if
8:    if q has target location then
9:       return "reachable"
10:   end if
11:   push(Q_reach, q)
12:   T_asap ← {⇒_d | q ⇒_d q' ∧ q' ∈ Q
                    ∧ ⇒_d has asap condition}
13:   if T_asap is not empty then
14:      for ⇒_asap ∈ T_asap do
15:         q ⇒_asap q'
16:         push(Queue_wait, q')
17:      end for
18:   else
19:      for q' ∈ {q_δ | q ⇒_δ q_δ} do
20:         push(Q_reach, q')
21:         for q'' ∈ {q'_d | q' ⇒_d q'_d ∨ q' ⇒_d* q'_d} do
22:            push(Q_wait, q'')
23:         end for
24:      end for
25:   end if
26: end while
27: return "not reachable"
```

表 1　実験環境
Table 1　Experiment environment.

OS	Gentoo Linux (Kernel 3.5.7, 64 bit)
CPU	Intel Core2 Quad Q9650 (3.00 GHz)
RAM	7,985 MB
検証器	Ruby 1.9.3-p363

なお，Q_δ は実数の稠密性により無限集合となる．この問題に対して，等価領域をまとめることで有限集合にする手法が提案されている [3], [5]．今回の実装では，Bengtsson らの手法 [5] を採用している．

4.3 検証実験

今回の検証実験では，検証対象とするオブジェクトに対してモニタオートマトンを埋め込み，モニタオートマトンに対する到達可能性解析でスケジューラビリティ検証を行う．今回の実験環境を**表 1** に示す．今回の実験で使用する検証器は Ruby で実装している．また，検証器内部での変数演算（一階述語論理）には数式処理システム Reduce [7] を使用している．

4.3.1 モニタオートマトン

検証のために使用するモニタオートマトンを**図 12** と**図 13** に示す．

図 12 の monitorDeadline は，各 CPU タスク（TaskA, TaskB）に埋め込まれるモニタオートマトンである．monitorDeadline では，CPU タスクの生成からの経過時間

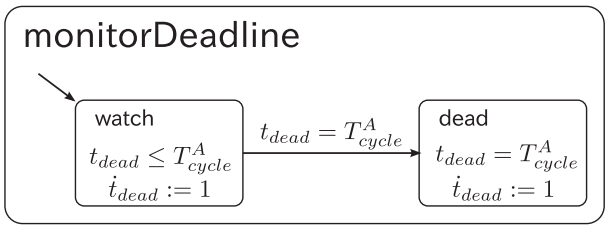

図 12　CPU タスクのデッドライン検知用モニタオートマトン
Fig. 12　Monitor automaton for detecting the deadline.

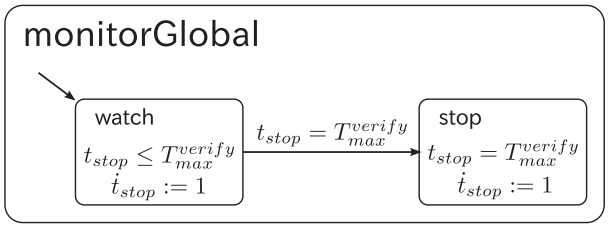

図 13　検証上限を設定するモニタオートマトン
Fig. 13　Monitor automaton to set the upper limit of verification.

を変数 t_{dead} で管理しており，デッドラインに到達した場合，ロケーション dead へ遷移する．スケジューラビリティ検証では，このロケーション dead への到達可能性を検証する．なお，変数 t_{dead} の初期値は 0 であり，各 CPU タスクのデッドラインは，それぞれの起動周期（T^A_{cycle}，T^B_{cycle}）とする．

図 13 は有限モデル検査で用いるオートマトンである．これはクラス Environment に埋め込まれる．モニタオートマトン monitorGlobal は検証開始からの経過時間を変数 t_{stop} で管理しており，検証上限 T^{verify}_{max} に達した場合，ロケーション stop に遷移する．そして，ロケーション stop の不変条件により変数 t_{stop} は T^{verify}_{max} より大きな値を保持できないため，検証は停止することになる．ここで，変数 t_{stop} の初期値は 0 である．また，ロケーション stop への遷移を確認することにより，初期状態から検証上限を含む状態へ至るパスが存在することを確認することができる．

今回の検証実験では，各 CPU タスクに埋め込まれたモニタオートマトン monitorDeadline のロケーション dead をターゲットロケーションとし，それらに対する到達可能性解析を行う．

4.3.2　実験 1：スケジューリング不可能例

スケジューリング不可能の例を扱う．検証時の各パラメータを**表 2** に示す．ここで，検証上限 $T^{verify}_{max} = 1800$ である．

表 2 のパラメータの場合，TaskB のモニタオートマトン monitorDeadline のロケーション dead へ到達するパスが存在する．このパラメータの場合，時刻 400 のときに TaskB の処理時間は TaskB.$e = 100$ となる．このとき，TaskB の処理終了まであと $T^B_{exe2} - \text{TaskB}.e = 10$ の時間が必要であるが，この時点で TaskB の生成からの経過時間 TaskB.$t_{dead} = 200$ となってしまう．そのため，TaskB は時間超過によりスケジューリング不可能となる．このときの演算時間は 4.92 秒，最大使用メモリは 1832 KB である．

表 2　スケジューリング不可能な場合のパラメータ
Table 2　Experiment parameters (unschedulable).

	TaskA (CoTaskA)		TaskB (CoTaskB)	
初期起動	T^A_{init}	70	T^B_{init}	200
起動周期	T^A_{cycle}	70	T^B_{cycle}	200
実行時間 1	T^A_{exe1}	20	T^B_{exe1}	0
実行時間 2	T^A_{exe2}	30	T^B_{exe2}	110
DRP 実行時間	T^{coA}_{exe}	20	T^{coB}_{exe}	5

表 3　スケジューリング可能な場合のパラメータ
Table 3　Experiment parameters (schedulable).

	TaskA (CoTaskA)		TaskB (CoTaskB)	
初期起動	T^A_{init}	70	T^B_{init}	200
起動周期	T^A_{cycle}	70	T^B_{cycle}	200
実行時間 1	T^A_{exe1}	20	T^B_{exe1}	0
実行時間 2	T^A_{exe2}	30	T^B_{exe2}	90
DRP 実行時間	T^{coA}_{exe}	20	T^{coB}_{exe}	5

4.3.3　実験 2：スケジューリング可能例

スケジューリング可能の例を扱う．検証時の各パラメータを**表 3** に示す．修正したパラメータでは，TaskB の実行時間 T^B_{exe2} を 110 から 90 に減らしている．ここで，検証上限 $T^{verify}_{max} = 1800$ である．

この条件では，CPU タスクに追加したモニタオートマトン monitorDeadline はロケーション dead へ到達しない．つまり，すべての起動した CPU タスクがデッドラインを超過せずに終了することが保証できる．このときの演算時間は 204.89 秒，最大使用メモリは 79,768 KB である．

5.　まとめ

本稿では，CPU と DRP が協調動作をする動的再構成可能システムを対象とした仕様記述言語の提案，および対象システムのモデル化と，モデル検査を用いた対象システムの安全性検証（スケジューラビリティ検証）を行った．本研究の主な貢献は次の 2 点である．

- 構造が動的に変更されるシステムを対象とした仕様記述言語を提案した．また，既存研究では表現が困難であった動的構造を，オブジェクト指向を取り入れることで簡潔に表現することを可能にした．
- 仕様記述したモデルに対して到達可能性解析を用いた検証が実行可能であることを示した．

本稿で提案した仕様記述言語では，オブジェクト指向を取り入れたため，既存研究よりも各クラス（オブジェクト）の結合度が低い．そのため，モデルに対する仕様の追加・変更が容易になっており，また，モデルの一部を他のモデルで再利用することが可能になっている．これらは，本稿で提案した仕様記述言語の利点である．

今後の課題として，本稿で提案した仕様記述言語を対象

とした検証器の改善や，検証手法の改良があげられる．本研究で実装した検証器は，モデル検査の基本的な検証手法である到達可能性解析しか行えない．そのため，述語論理式を用いたより高度な検証を行うために，検証器を改善する必要がある．また，モデル検査は一般的に状態爆発（探索空間の指数的増加）が発生するため，それを回避する検証手法が必要となる．ハイブリッドオートマトンに対しては，すでに記号モデル検査 [8] や CEGAR を用いた手法 [2] によって状態爆発抑制の研究が行われている．これら手法を本稿で提案した言語へ拡張することにより，状態爆発を回避しながら，より高度な検証が可能になると考えられる．

参考文献

[1] Alur, R., Courcoubetis, C., Henzinger, T.A. and Ho, P.-H.: Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems, *Hybrid Systems*, Lecture Notes in Computer Science, Vol.736, pp.209–229, Springer-Verlag (1993).

[2] Alur, R., Dang, T. and Ivančić, F.: Predicate Abstraction for Reachability Analysis of Hybrid Systems, *TECS*, Vol.5, No.1, pp.152–199 (2006).

[3] Alur, R. and Dill, D.L.: A Theory of Timed Automata, *Theoretical Computer Science*, Vol.126, No.2, pp.183–235 (1994).

[4] Attie, P.C. and Lynch, N.A.: Dynamic Input/Output Automata: A Formal Model for Dynamic Systems, LNCS, Vol.2154, pp.137–151 (2001).

[5] Bengtsson, J. and Yi, W.: On Clock Difference Constraints and Termination in Reachability Analysis of Timed Automata, *Formal Methods and Software Engineering*, Lecture Notes in Computer Science, Vol.2885, pp.491–503, Springer Berlin Heidelberg (2003).

[6] Deshpande, A., Gollu, A. and Semenzato, L.: The SHIFT Programming Language for Dynamic Networks of Hybrid Automata, *IEEE Trans. Automatic Control*, Vol.43, No.4, pp.584–587 (1998).

[7] Hearn, A.C.: REDUCE Computer Algebra System, (online), available from ⟨http://reduce-algebra.com/⟩ (accessed 2013-01-20).

[8] Henzinger, T.A., Ho, P.-H. and Wong-toi, H.: HyTech: A Model Checker for Hybrid Systems, *Software Tools for Technology Transfer*, Vol.1, pp.460–463 (1997).

[9] Hoare, C.A.R.: Communicating sequential processes, *Comm. ACM*, Vol.21, No.8, pp.666–677 (1978).

[10] Kratz, F., Sokolsky, O., Pappas, G. and Lee, I.: R-Charon, a Modeling Language for Reconfigurable Hybrid Systems, LNCS, Vol.3927, pp.392–406 (2006).

[11] Onogi, K. and Ushio, T.: Scheduling of Periodic Tasks on a Dynamically Reconfigurable Device Using Timed Discrete Event Systems, *IEICE trans.*, Vol.89, No.11, pp.3227–3234 (2006).

[12] Teich, J. and Köster, M.: (Self-)reconfigurable Finite State Machines: Theory and Implementation, *Proc. conference on Design, automation and test in Europe, DATE '02*, Washington, DC, USA, pp.559–568, IEEE Computer Society (2002).

[13] Tuan, V.M., Hasegawa, Y., Katsura, N. and Amano, H.: Performance Evaluation of Hardware Multi-process Execution on the Dynamically Reconfigurable Processor, IEICE Technical Report, Vol.106, No.247, pp.25–30 (2006).

[14] 天野英晴，安達義則，堤　聡，石川健一郎：動的リコンフィギャブルプロセッサにおける可変クロック機構の導入，情報処理学会研究報告，Vol.2005, No.8, pp.13–16 (2005).

[15] 瀧内新悟，山根　智：事例研究：プリエンプティブな周期タスクからなる組込みソフトウェアのモデリング，仕様記述と有限モデル検査，電子情報通信学会論文誌，Vol.93, No.11, pp.2403–2415 (2010).

[16] 山崎真一，山根　智，安井雅俊：階層時間オートマトン群の並列動作の述語抽象化精錬検証手法，電子情報通信学会技術研究報告，Vol.108, No.278, pp.31–36 (2008).

[17] 本村真人，藤井太郎，古田浩一朗，安生健一朗，矢部義一，戸川勝巳，山田順也，伊澤義貴，佐々木僚子：4. 動的再構成プロセッサ（DRP）（実例，＜特集＞新世代マイクロプロセッサアーキテクチャ（後編）），情報処理学会誌，Vol.46, No.11, pp.1259–1265 (2005).

[18] 長谷川揚平，阿部昌平，黒瀧俊輔，ヴマントウアン，天野英晴：動的リコンフィギャラブルプロセッサにおける時分割多重実行の評価（リコンフィギャラブルシステム），情報処理学会論文誌，Vol.47, No.12, pp.171–181 (2006).

[19] 中橋　亮，木谷友哉，安本慶一，中田明夫，東野輝夫：リアルタイム組込みシステムの動的再構成可能プロセッサへの一実装方法の提案，IEICE technical report, Vol.105, No.450, pp.31–36 (2005).

[20] 南　翔太，瀧内新悟，瀬古口智，中居祐輝，山根　智：動的再構成可能プロセッサのモデル化，仕様記述とモデル検査，コンピュータソフトウェア，Vol.28, No.1, pp.190–216 (2010).

山田 英史

2011 年金沢大学工学部卒業．同年金沢大学大学院自然科学研究科電子情報工学専攻入学，現在も在学．動的再構成可能プロセッサのモデル化と仕様記述言語に関する研究に従事．現在，NEC 勤務．

中居 祐輝

2010 年金沢大学大学院自然科学研究科電子情報工学専攻修士課程修了．在学中は動的再構成可能プロセッサのモデル化と仕様記述言語に関する研究に従事．現在，ソニー勤務．

山根 智　（正会員）

1984 年京都大学大学院修了．現在，金沢大学理工研究域電子情報学系教授．博士（京都大学）．リアルタイムハイブリッドシステム等の形式的検証の研究に従事．EATCS，日本ソフトウェア科学会等各会員．