時間オートマトンを対象とした コンポーネント部分合成による抽象化洗練手法の改良

長岡武志[†] 岡野浩三[†] 楠本真二[†]

著者らの提案している時間オートマトンの抽象化精錬手法では，時間オートマトンが持つクロック変数をすべて除去するという抽象化を行うため，生成する抽象モデルは有限状態オートマトンとなる．これまでの手法では，複数のプロセスが並列に動作するようなモデルを検査する場合，事前に複数のプロセスを並列合成する必要があった．しかし，事前に合成を行ったモデルに対して検査を行う場合，モデル検査実行時に合成を行う場合に比べ，消費するメモリ量が増加し，抽象化の効率が低下してしまう．そこで本稿ではそのようなプロセス合成をインクリメンタルに行うことにより，部分的にとどめるような手法を提案する．また簡単な例題に対する予備実験の結果を報告する．

Improvement of Abstraction Refinement for Timed Automata based on Partial Parallel Composition

TAKESHI NAGAOKA,[†] KOZO OKANO [†] and SHINJI KUSUMOTO[†]

We have proposed an abstraction refinement technique for timed automata. In our former abstraction technique, we perform abstraction by removing all clock variables from timed automata. Therefore, generated abstract models are finite automata. In the method, a network of parallel timed automata should be composed into a single timed automaton, as preliminary of model checking. Such an approach, however, often yeilds to memory consumption; which decreases the efficiency. This paper improves our abstraction technique not to need full parallel composition, by incremental construction technique. The result of a small experiment is also described.

1. まえがき

モデル検査手法はシステムを有限の状態遷移系として記述し，状態遷移系の全状態を探索することで，システムが仕様を満たすかどうかを証明する手法である．しかし，大規模なシステムに対しては状態数爆発を起こすなど，スケーラビリティの弱さが課題となっている．モデル検査のスケーラビリティの弱さを改善する手法として，検査する性質ごとに，モデルの状態数を適切に削減するモデル抽象化手法が注目されている[1),2)]．

一方，実時間システムの動作検証には，有限の状態遷移系に実時間制約を付加した時間オートマトン[5)]が用いられる．時間オートマトンでは，有限のロケーションと呼ばれる状態に，実数値をとるクロック変数を用いた制約が付加されるため，時間オートマトンは無限の状態空間を持つことになる．モデル抽象化を行わない従来の時間モデル検査では時間領域が実質，有限個に押さえられることを利用し有限状態のモデルに対し検査を行う．しかし，この状態数はロケーションやクロックの個数に対して指数的に増加する．したがって時間オートマトンに対しても，状態数を削減するための適切な抽象化の方法が必要となる．著者らは文献 9)，10) で，時間オートマトンを対象としたモデル抽象化手法を提案している．9)，10) で提案している手法では，抽象化を適用したモデル（抽象モデル）上で発生した反例を利用して抽象モデルの洗練を行う，CounterExample-Guided Abstraction Refinement Loop(CEGAR Loop)[1)] の枠組みを利用しており，抽象化の全工程を自動で行うことができる．また，提案している手法では，時間オートマトンが持つクロック変数をすべて削除し，時間による振る舞いの違いをモデルの遷移関係を操作することで抽象モデルを洗練していく．具体的には，状態，遷移の複製，遷移の除去などの操作を行うことで実現している．しかしこれまでの手法では，複数のプロセスが並列に動作するようなモデルを検査する場合，事前に複数のプロセスを並列合

† 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University

成する必要があった．このように事前に合成を行ったモデルに対して検査を行った場合，モデル検査実行時に合成を行う場合に比べ，消費するメモリ量が増加し，抽象化の効率が低下してしまうという問題が生じる．

本稿では，このような問題を解決するため，これまでの手法を，モデルが持つ並列プロセスの合成が必要とならないよう改善する．具体的には，抽象モデル洗練の際に，状態，遷移の複製，遷移の削除などの操作が必要な部分について部分的に並列合成を行うことで，モデル全体を合成することを回避している．このとき，部分的に合成したプロセスの動作とその他のプロセスの動作を同期させることで，もとの時間オートマトンと等価な動きをすることを保証する．その際，モデル検査器 UPPAAL[6),7)] で用いられる拡張時間オートマトンの構成要素である整数変数や Broadcast Channel, Committed Location などを利用している．

以降，2 では準備として時間オートマトンの定義を与え，一般的な CEGAR ループ，著者らが提案している抽象化アルゴリズムを説明する．3 では，2 で説明した既存のアルゴリズムを並列合成が必要とならないように改善する手法を述べる．さらに 4 で実験結果を示し，最後に，5 でまとめる．

2. 準　　備

本章では準備として一般的な時間オートマトン，ネットワーク時間オートマトンの定義を与え，UPPAAL 拡張時間オートマトン，著者らがこれまでに提案してきた抽象化手法について説明する．

2.1 一般的な時間オートマトン

時間オートマトンは有限状態のオートマトンに時間経過を表現するクロック変数を付加したオートマトンである．

定義 2.1 (時間オートマトン)**.** 時間オートマトン $\mathscr{A} = (L, l_0, T, I, C, A)$，$C$: クロックの有限集合;，A : アクションの有限集合;，L : ロケーションの有限集合;，$l_0 \in L$:初期ロケーション;，$T \subset L \times A \times 2^{c(C)} \times \mathscr{R} \times L$;，ここで, $2^{c(C)}$: ガード群;，$\mathscr{R} = 2^C$: リセットクロック群;，$I \subset (L \to 2^{c(C)})$: インバリアント付加関数.

遷移 $t = (l_1, a, g, r, l_2) \in T$ は $l_1 \xrightarrow{a,g,r} l_2$ と表記する．$\nu : C \to \mathbb{R}_{\geq 0}$ はクロックに値を割り当てる関数である．さらに，ν のドメインを $\nu \in \mathbb{R}^C_{\geq 0}$ と拡張することができる．$d \in \mathbb{R}_{\geq 0}$ に対して，$(\nu+d)(x) = \nu(x)+d$ とし，$r \in 2^C$ に対して，$r(\nu) = \nu[x \mapsto 0], x \in r$ とする．また ν 全体の集合を N と表記する．

さらに，時間オートマトンの意味は次のように与えることができる．

定義 2.2 (時間オートマトンの意味)**.** 時間オートマトン $\mathscr{A} = (L, l_0, T, I, C, A)$ に対して
$\mathscr{A}$ の状態集合を $S = L \times N$ とする．
$\mathscr{A}$ の初期状態は $(l_0, 0^C) \in S$ で与えられる．
状態遷移 $l_1 \xrightarrow{a,g,r} l_2$ $(\in T)$ に対し，次の 2 つの遷移が定義される．

$$\frac{l_1 \xrightarrow{a,g,r} l_2, g(\nu), I(l_2)(r(\nu))}{(l_1, \nu) \xrightarrow{a} (l_2, r(\nu))}$$

$$\frac{\forall d' \leq d \quad I(l_1)(\nu + d')}{(l_1, \nu) \xrightarrow{d} (l_1, \nu + d)}$$

前者をイベント遷移, 後者を時間遷移と呼ぶ.

定義 2.3 (DBM(Difference Bound Matrix))**.** 時間オートマトン $\mathscr{A} = (L, l_0, T, I, C, A)$ に対して *DBM* は $|C|$ -次元ユークリッド空間上の凸空間を表現し，N の要素の集合として表現される．*DBM* D に対して，$(l, D) = \{(l, \nu) | \nu \in D\}$ と表記する．

2.2 ネットワーク時間オートマトン

時間オートマトン間の並列実行，同期化を実現するため，オートマトン間の同期を行う半アクション (チャネル) が用いられる[5)]．なお，半アクションに対して，単一プロセス内のみで行われるアクションを全アクションとする．時間オートマトン $\mathscr{A}_1, \ldots, \mathscr{A}_n$ に対して並列合成モデル $\mathscr{A} := \mathscr{A}_1 || \mathscr{A}_2 || \cdots || \mathscr{A}_n$ はネットワーク時間オートマトンである．

定義 2.4 (ネットワーク時間オートマトンの意味)**.** ネットワーク時間オートマトン $\mathscr{A} := \mathscr{A}_1 || \mathscr{A}_2 || \cdots || \mathscr{A}_n$ に対して遷移システム $(S, s_0, \to)$ が定義される．$(\vec{l}, \nu) \in S$ に対して，$\vec{l} = (l_1, \ldots, l_n)$ *(l_i は $\mathscr{A}_i$ 上のロケーション)* とする．$C = \bigcup_{1 \leq i \leq n} C_i$ としたとき，$\nu \in \mathbb{R}^C_{\geq 0}$ とする．また $s_0 = (\vec{l_0}, 0^C)$ *($\vec{l_0} = (l_{1,0}, \ldots, l_{n,0})$)* である．$\vec{l}[l'_i/l_i]$ は $\vec{l}$ の l_i を l'_i に置き換えたベクトルを表す．ネットワーク時間オートマトンの遷移 $\to$ は次のように定義される．

- $\mathscr{A}_i$ の全アクション a に対して,
  $(\vec{l}, \nu) \xrightarrow{a} (\vec{l}[l'_i/l_i], r(\nu))$
  `if` $l_i \xrightarrow{a,g,r} l'_i$ $(\in T_i)$ `and` $g(\nu)$
- $\mathscr{A}_i$, $\mathscr{A}_j$ *($i \neq j$)* の半アクション $x!$, $x?$ に対して,
  $(\vec{l}, \nu) \xrightarrow{x} (\vec{l}[l'_i/l_i, l'_j/l_j], r_i \cup r_j(\nu))$
  `if` $l_i \xrightarrow{x!,g_i,r_i} l'_i$ $(\in T_i)$ `and` $l_j \xrightarrow{x?,g_j,r_j} l'_j$ $(\in T_j)$ `and` $g_i(\nu)$ `and` $g_j(\nu)$
- $d \in \mathbb{R}_{\geq 0}$ に対して
  $(\vec{l}, \nu) \xrightarrow{d} (\vec{l}, \nu + d)$, `if` $\forall i, \forall d' \leq d,\ I(l_i)(\nu + d')$

DBM D を用いて S の部分集合を $(\vec{l}, D) \in \{(\vec{l}, \nu) | \nu \in D\}$ と表記する．

2.3 UPPAAL 拡張時間オートマトン

時間モデル検査器である UPPAAL では，標準的な時

間オートマトンに対して拡張を行った拡張時間オートマトンを検証モデルとしている[7]．本節では，UPPAAL 拡張時間オートマトンの構成要素の中で，本稿で利用する整数変数，Broadcast Cannel，ロケーションについて説明する．

2.3.1 整 数 変 数

整数変数は，インバリアントやガード制約，変数の代入文に含めることが可能であり，ロケーションの遷移時に値を更新することが可能である．

2.3.2 Broadcast Channel

本稿では，2 プロセス間の同期を行う Binary Channel に加え，現状態で同期可能なプロセスすべてと同期を行う Broadcast Channel を用いる．

Broadcast Channel として宣言されたチャネル c は，$c!$ とラベル付けされた送信側の遷移に対して，現状態から実行可能なすべての受信側の遷移 ($c?$ とラベル付けされた遷移) と同期する．ただし，$c!$ とラベル付けされた遷移に対して，現状態から実行可能な $c?$ とラベル付けされた遷移が存在しなければ、$c!$ とラベル付けされた遷移が他の遷移と同期することなく単独で実行される．

2.3.3 Committed Location

Committed Location はそのロケーション上での時間経過が不可能であり，あるプロセスが Committed Location に遷移した場合は，次に起こるアクション遷移は必ず Committed Location から抜け出す遷移でなければならない．また，同様に時間経過を許さないロケーションとして Urgent Location がある．しかしあるプロセスが Urgent Location へ遷移した場合，そのプロセスが Urgent Location から抜け出す遷移の前に他のプロセスのアクション遷移が割り込むことが許されている．

2.4 一般的な CEGAR ループ

モデル抽象化では，過剰な抽象化や誤った抽象化によって本来のモデルでは発生するはずのない見せかけの反例が発生してしまう場合がある．文献 1) では，最初に十分な抽象化を行い，抽象モデル上で見せかけの反例が発生すればその反例が発生しないように抽象化されたモデルを洗練する，CEGAR(Counterexample-Guided Abstraction Refinement) アルゴリズムを提案している．一般的な CEGAR アルゴリズムを図 1 に示す．

一般的な CEGAR ループでは，まず最初に十分な抽象化を行う初期抽象化を行う．次に抽象化を適用したモデル (抽象モデル) に対してモデル検査を適用する．このとき，反例が検出された場合は，その反例が本来のモデル (具体モデル) 上で実行可能かどうかを調べるシミュレーションを行う．さらに，反例が具体モデル上で実行不可能である (そのような反例を偽反例と呼ぶ) 場合は，偽反例が生じないよう抽象モデルの洗練を行い，再びモデル検査を適用する．CEGAR ループでは以上の流れを正しい検査結果が得られるまで繰り返す．

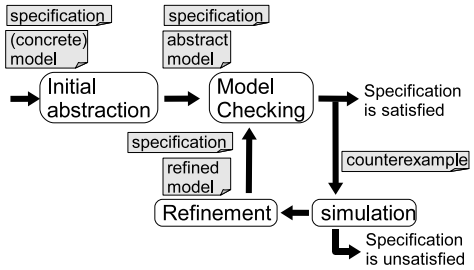

図 1 CEGAR ループ

2.5 これまでに提案してきた抽象化手法

本節では，文献[9),10)] でこれまでに提案してきた抽象化アルゴリズムを示す．提案してきた抽象化はシステムの異常状態への到達可能性に関する検証を目的としている．

まず初期抽象化によって，抽象化関数 h によって時間オートマトン $\mathscr{A}$ から抽象モデル $\hat{M}$ を生成し，$\hat{M}$ に対してモデル検査を行う．モデル検査の結果，反例 $\hat{T}$ が発生した場合は，$\hat{T}$ の系列を h の逆関数 h^{-1} を用いて具体化する．この $\hat{T}$ を具体化した $\mathscr{A}$ 上の実行系列の集合を T とする．シミュレーションでは T の各要素について DBM の操作関数を用いて $\mathscr{A}$ 上で実行可能かどうかを調べる．T のすべての要素が実行不可能であれば，反例 $\hat{T}$ が実行不可能となるように改良を行わなければならない．提案してきた抽象化アルゴリズムでは，改良の際に直接 $\hat{M}$ の変形を行うのではなく，$\mathscr{A}$ に対して等価変形を行い，変形後の時間オートマトンに対して再び抽象化関数 h を適用し，新しい抽象モデルを生成する．以上の提案アルゴリズムによる CEGAR ループの流れを図 2 に示す．

2.5.1 抽象モデル

時間オートマトン $\mathscr{A}$ によって生成される抽象モデル $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\rightarrow})$ は $\mathscr{A}$ からクロック変数をすべて削除した状態遷移モデルであり，同時に，遷移のラベルやロケーションのインバリアントからクロック変数に関する部分が除去されている．したがって，抽象化関数 $h : L \rightarrow \hat{S}$ は定義 2.5 のように与えられる．

定義 2.5 (抽象化関数 h)**.** 時間オートマトン $\mathscr{A} = (L, l_0, T, I, C, A)$ に対し，抽象化関数 $h : L \rightarrow \hat{S}$ を以下のように定義する．

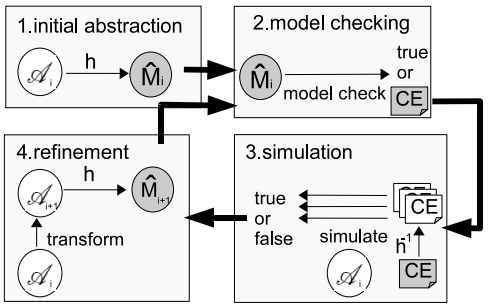

図 2 これまでに提案している抽象化

- $\forall l_i, l_j \in L.\ h(l_i) = h(l_j) \iff l_i = l_j$

同様に抽象化関数 h の逆関数として $h^{-1} : \hat{S} \to L$ も定義される．

定義 2.5 の抽象化関数 h を用いて生成される，時間オートマトン $\mathscr{A}$ に対する抽象モデル $\hat{M}$ を定義 2.6 で与える．

定義 2.6 (抽象モデル)**.** 時間オートマトン $\mathscr{A} = (L, l_0, T, I, C, A)$ を定義 *2.5* の抽象化関数 h によって抽象化したモデル $\hat{M} = (\hat{S}, \hat{s}_0, \hat{\to})$ を以下のように与える．

- $\hat{S} = \{h(l) | l \in L)\}$
- $\hat{s_0} = h(l_0)$
- $\hat{\to} = \{(\hat{l_1}, a, \hat{l_2}) | (l_1, a, g, r, l_2) \in T \wedge \hat{l}_1 = h(l_1) \wedge \hat{l}_2 = h(l_2)\}$

定義 2.7 (反例)**.** $\hat{M}$ 上の反例は抽象状態 $\hat{S}$ の系列であり，長さ n の反例 $\hat{T}$ は $\hat{T} = \langle \hat{s}_0, \cdots, \hat{s}_n \rangle$ と表記する．

反例 $\hat{T} = \langle \hat{s}_0, \cdots, \hat{s}_n \rangle$ に対し，$\mathscr{A}$ 上で $\hat{T}$ に対応する遷移の系列の集合 T は抽象化関数の逆関数 h^{-1} を用いて以下のように求められる．

$$
\begin{aligned}
T = \{(l_0 \xrightarrow{a_1, g_1, r_1} l_1 \xrightarrow{a_2, g_2, r_2} \cdots \xrightarrow{a_n, g_n, r_n} l_n) | \\
(l_i = h^{-1}(s_i) \ \texttt{for}\ 0 \le i \le n\ ) \wedge \\
((l_{i-1}, a_i, g_i, r_i, l_i) \in T \ \texttt{for}\ 1 \le i \le n\ )\}
\end{aligned}
$$

2.5.2 初期抽象化

初期抽象化では時間オートマトン $\mathscr{A} = (L, l_0, T, I, C, A)$ に対して抽象化関数 h を適用し，抽象モデル $\hat{M}$ を生成する．

2.5.3 シミュレーション

シミュレーションでは，反例 $\hat{T}$ に対応する $\mathscr{A}$ 上の遷移の系列の集合 T に対し，T の各の系列 t についてシミュレーションを行う．シミュレーションでは時間オートマトンのクロックに関する状態空間を操作するためのデータ構造 DBM[5] を利用し，t の先頭のロケーションから最後のロケーションまで到達可能であるかどうかを調べる．

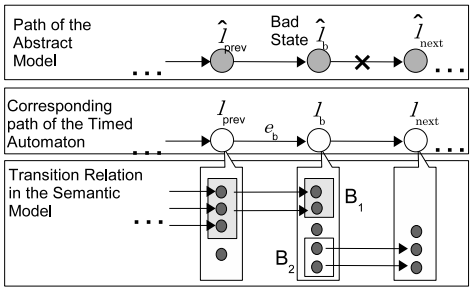

図 3 反例

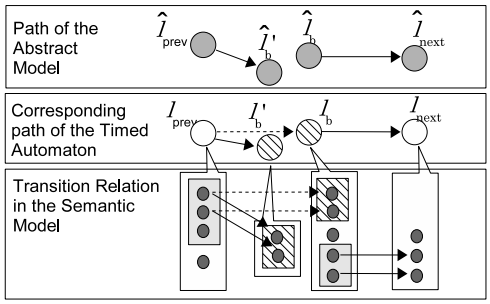

図 4 洗練後のモデル

2.5.4 抽象モデル洗練

抽象モデル上の反例が偽反例である場合，図 3 のように，初期状態 $(l_0, 0^C)$ からは到達可能であるが次のロケーション l_{next} への遷移が不可能であるような状態の集合 $B_1 = (l_b, D_1)$ と，$(l_0, 0^C)$ からは到達不可能であるが l_{next} への遷移が可能であるような状態の集合 $B_2 = (l_b, D_2)$ の両方が縮約されている抽象状態 $\hat{l}_b$ が存在する．このとき $\hat{l}_b$ を Bad State と呼ぶこととする．このような場合，本来 $(l_0, 0^C)$ からは到達不可能な状態集合 B_2 からの遷移によって，偽反例が生じてしまう．したがって，B_2 に対応する抽象状態へ偽反例のパスでは到達不可能となるように洗練をおこなう必要がある．提案アルゴリズムでは，具体モデル上で B_1 に対応するロケーション l'_b を複製し，l_{prev} から l_b への遷移を削除することによって，抽象モデルにおいても B_2 が縮約されている抽象状態へ到達不能となるように洗練を行う．図 4 は洗練後のモデルを示している．図 4 の点線で描かれた遷移は洗練によって削除された遷移を表している．なお B_1 はシミュレーション時に同時に求めることができる．

2.5.5 提案してきた手法の問題点

これまでに提案してきた抽象化アルゴリズムをネットワーク時間オートマトンに対して適用する場合，抽象モデルの洗練の際に，状態集合 $(\vec{l}, D)$ の複製や遷移 $(\vec{l}, \nu) \stackrel{a}{\Rightarrow} (\vec{l}[l'_i/l_i], r(\nu))$ の削除などの操作が必要と

なる．このとき，洗練操作の対象はネットワーク時間オートマトンの意味モデルとなっている．したがって，ネットワーク時間オートマトンに対して既存の抽象化を適用するためには，事前にネットワーク時間オートマトンのすべてのプロセスを並列合成したモデルを記述する必要がある．このように事前にプロセスを並列合成したモデルをモデル検査器に入力すると，モデル検査の際のメモリ消費量が大幅に増加してしまい，抽象化の効率を低下させてしまうことが問題となっている．これは，事前に並列合成を行うことにより，Partial Order Reduction[8] など，モデル検査時に並列プロセスの状態探索を効率的に行うテクニックが活用できないことが原因の一つとして挙げられる．したがって，並列プロセスを合成せずに，別々のプロセスとして表現したまま洗練操作が適用可能となるように洗練操作を改良する必要がある．

3. 提案する洗練手法

本章では，2.5 節で述べた既存の抽象化手法の改善手法を述べる，これまでの手法では，ネットワーク時間オートマトンを入力とした場合，予め並列プロセスの合成を行い一つのプロセスとして表現した後，抽象化を適用していた．改善手法では，事前の並列合成を行わず，別々のプロセスとして表現したまま抽象化を適用する．

提案手法の基本的なアイデアとしては，モデル検査，シミュレーションによって検出した偽反例を取り除くために必要となる部分に関して，部分的な合成モデルを生成し，部分的に合成したプロセスの動作とその他のプロセスの動作を同期させることで，本来は実行不可能であるような動作を制限する．このとき部分的に合成したモデルを $\mathscr{A}_{com}$ と表記する．

提案手法では，ネットワーク時間オートマトン $\mathscr{A} = \mathscr{A}_1||\cdots||\mathscr{A}_n$ に対して，部分的な合成モデル $\mathscr{A}_{com}$ を追加した $\mathscr{A}' = \mathscr{A}_1||\cdots||\mathscr{A}_n||\mathscr{A}_{com}$ に対して初期抽象化，モデル検査，シミュレーション，モデルの洗練操作を適用する．

提案手法では，$\mathscr{A}_{com}$ が他のプロセスの動作を制御するために，新たに大域整数変数を追加している．また，チャネルや Committed Location を利用して $\mathscr{A}_{com}$ が他のプロセスの遷移と同期して動作するようにしている．

3.1 管 理 変 数

各プロセスの現状態と，直前に実行した遷移を管理するために，各プロセスごとに 2 つの大域整数変数を追加する．したがって n 個のプロセスを持つモデルでは，$2n$ 個の大域変数を新たに追加する．$\mathscr{A}_i$ の現状態を管理する整数を $stat_i$，直前に実行した遷移を管理する変数を tr_i と表記する．各プロセスの各ロケーション，遷移には固有の ID を割り当て (l の ID を $Id_{loc}(l)$，遷移 t の ID を $Id_{tr}(t)$ とする)，$\mathscr{A}_i$ の各遷移 $t_i = (l_1, a, g, r, l_2)$ について，$stat_i := Id_{loc}(l_2)$，$tr_i := Id_{tr}(t_i)$ という代入文を付加する．このようにすることで，$\mathscr{A}_i$ の現在のロケーションは $Id_{loc}(l) = stat_i$ であるようなロケーションであり，直前に実行した遷移は $Id_{tr}(t) = tr_i$ であるような遷移であることが他プロセスからも把握することができる．

また，同様に $\mathscr{A}_{com}$ の管理変数 $stat_{com}$，tr_{com} も追加する．

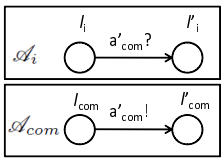

図 5 独立な遷移の同期

3.2 他のプロセスの遷移との同期

本節では $\mathscr{A}_{com}$ と他のプロセスとの同期方法を述べる．ここでは，$\mathscr{A}_{com}$ の遷移 $t_{com} = l_{com} \xrightarrow{a,g,r} l'_{com}$ を他のプロセスの遷移と同期させることを考える．このとき，同期させる遷移が，全アクションによる遷移 (単一プロセス内で行われる遷移) である場合と，半アクションによる遷移 (他のプロセスの遷移と同期する遷移) である場合のそれぞれについて説明する．

i) 全アクションによる遷移

$\mathscr{A}_i$ の独立な遷移 $t_i = l_i \xrightarrow{a_i,g_i,r_i} l'_i$ と同期させる場合，新たな 2 プロセス間のチャネル a'_{com} を生成し，a'_{com} を通じて t_{com} と t_i を同期させる (図 5).

ii) 半アクションによる遷移

t_{com} と同期させる遷移を $t_i = l_i \xrightarrow{a!,g_i,r_i} l'_i$ $t_j = l_j \xrightarrow{a?,g_j,r_j} l'_j$ $(i \neq j)$ とする．同一の遷移で複数のアクションを実行できないため，$\mathscr{A}_i$，$\mathscr{A}_j$ にそれぞれ新たなロケーション l''_i，l''_j を生成し，新たに遷移 $t'_i = l_i \xrightarrow{a'_{com},g_i,r_i} l''_i$，$t'_j = l_j \xrightarrow{a'_{com},g_j,r_j} l''_i$ を追加する．このとき a'_{com} は Broadcast Channel であり，遷移 t_{com} との同期を行う．さらにそれぞれ $t''_i = l''_i \xrightarrow{a!,g_i,r_i} l'_i$，$t''_j = l''_j \xrightarrow{a?,g_j,r_j} l'_j$ を生成し $\mathscr{A}_i$ と $\mathscr{A}_j$ の遷移を同期させる．また，これらの遷移が連続して実行されるように，生成するロケーション l''_i，l''_j は Committed Location とする．図 6 は $\mathscr{A}_{com}$ と同期させる遷移が互いに同期する遷移である場合の図である．c と書かれたロ

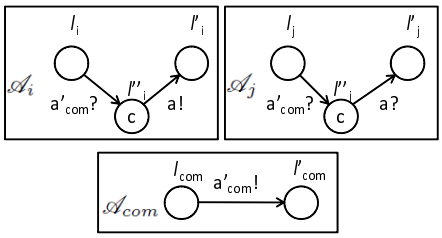

図 6 互いに同期する遷移との同期

ケーションは Committed Location を表している．

3.3 部分的な合成モデル $\mathscr{A}_{com}$

提案手法では，抽象モデル上の偽反例を除去するため，部分的な並列合成モデル $\mathscr{A}_{com}$ を生成し，モデル全体の動作を制限する．モデルの動作を制限する必要があるのは偽反例に関する部分であり，Bad State である $\vec{l}_b$ への遷移と $\vec{l}_b$ からの遷移を制限する必要がある．したがって，提案手法では抽象モデルの洗練の際に，$(\vec{l}_b, D_1)$ を表現する合成ロケーション $l_{com,b}$ を追加していく．

$\mathscr{A}_{com}$ はクロック変数を持たないプロセスであり，$\mathscr{A}_{com}$ のロケーション集合 L_{com} は以下の要素から構成される．

- 初期状態の集合 $(\vec{l}_0, D_0)$ ($(\vec{l}_0, 0^C)$ から時間遷移のみによって到達可能な状態集合) が合成されたロケーション $l_{com,0}$
- 状態の複製が適用される状態集合 $B_1 = (\vec{l}_b, D_1)$ を表現する合成ロケーション
- 上記以外の状態集合を表すロケーション l_{other}

$\mathscr{A}_{com}$ では，偽反例を除去するために状態の複製を適用する状態 $B_1 = (\vec{l}_b, D_1)$ に対する合成ロケーションが追加されていく．しかし，それ以外の状態をすべて l_{other} として表すことで，$\mathscr{A}$ のすべての状態を合成することを回避している．つまり，$\mathscr{A}_{com}$ は偽反例を取り除くために必要な部分のみが合成されているような合成モデルであると言える．

3.4 提案手法による CEGAR ループ

提案手法による抽象化方法について各ステップごとに説明する．前述のとおり，提案手法では，ネットワーク時間オートマトン $\mathscr{A} = \mathscr{A}_1 || \cdots || \mathscr{A}_n$ に対して，部分的な合成モデル $\mathscr{A}_{com}$ を追加した $\mathscr{A}' = \mathscr{A} || \mathscr{A}_{com}$ に対して初期抽象化，モデル検査，シミュレーション，モデルの洗練操作を適用する．以降では，記号の右肩に ‘ ′ ’(プライム) を添えた場合，$\mathscr{A}$ に $\mathscr{A}_{com}$ を追加した $\mathscr{A}'$ の要素を表し，右下に ‘com’ を添えた記号は A_{com} の要素を表している．

3.4.1 前 処 理

提案手法による CEGAR ループを実行する前処理として，$\mathscr{A}_{com}$ を生成する．$\mathscr{A}_{com}$ のロケーション集合 L_{com} は $\mathscr{A}$ の初期状態の集合 $(\vec{l}_0, D_0)$ を表す $l_{com,0}$ とそれ以外の状態を表現する l_{other} のみから構成される．また，$(\vec{l}_0, D_0)$ から実行可能なアクション遷移はすべて $\mathscr{A}_{com}$ の $l_{com,0}$ から l_{other} への遷移と同期させる必要がある．

3.4.2 初期抽象化

$\mathscr{A}$ の個々の時間オートマトンから，クロック変数，クロック変数に関する制約式やリセット式などを削除する．このとき，整数変数やチャネルは保持する．また，$\mathscr{A}_{com}$ はクロック変数を持たず，他のクロック変数を参照することもないため，$\mathscr{A}_{com}$ には処理は行わない．

また，生成される抽象モデルはクロック変数を持たない有限オートマトンのネットワークとして表現することができる．従って抽象モデルの状態は各有限オートマトンの状態の積として表現可能である．以降ではネットワーク時間オートマトンのロケーションベクトル $\vec{l}$ に対応する抽象状態を $\hat{\vec{l}}$ のように表現する．

3.4.3 モデル検査

$\mathscr{A}'$ に対して初期抽象化を行ったモデルに対してモデル検査器を用いて検証を行う．

3.4.4 シミュレーション

モデル検査によって検出された反例 $\hat{T}$ について対応する $\mathscr{A}'$ 上の遷移の系列 T を求め，T に含まれるすべての系列についてシミュレーションを行う．各系列のシミュレーションはネットワーク時間オートマトンの意味モデルに従い DBM の操作関数を適用することで，実現することができる．

3.4.5 抽象モデル洗練

抽象モデルの洗練では，シミュレーションによって得られる $B_1 = (\vec{l}_b, D_1)$，$\vec{l}_b$ の前後のロケーション $\vec{l}_{prev}$，$\vec{l}_{next}$ を用いる．$\vec{l}_b$，$\vec{l}_{prev}$，$\vec{l}_{next}$ において $\mathscr{A}$ のロケーションベクトルをそれぞれ $\vec{l}_b$，$\vec{l}_{prev}$，$\vec{l}_{next}$，$\mathscr{A}_{com}$ のロケーションをそれぞれ $l_{com,b}$，$l_{com,prev}$，$l_{com,next}$ と表すこととする．

抽象モデルの洗練手法は，状態の複製，遷移の複製，遷移の除去の操作から構成されている．状態の複製，遷移の複製，遷移の除去のアルゴリズムはそれぞれ図 7，8，9 に示している．なお，アルゴリズム中では記述を簡単化するため，ネットワーク時間オートマトン $\mathscr{A} = \mathscr{A}_1 || \ldots || \mathscr{A}_n$ を仮想的に合成した $\mathscr{A} = (L, l_0, T, I, C, A)$ という表現を用いている．またネットワーク時間オートマトン上の非同期の遷移ま

DuplicateState
Input $\mathscr{A}_{com}, B_1 = (\vec{l}'_b, D_1)$

$\{\mathscr{A}_{com} = (L_{com}, l_{com,0}, T_{com}, I_{com}, C_{com}, A_{com})\}$
$l_{com,new} := newLoc()$ {Generate a new location l'_b}
$L_{com} := L_{com} \cup \{l_{com,new}\}$
$I(\ l_{com,new}\ ) := Invariant(D_1)$
{A set of inequalities representing D_1}
$I(l_{com,new}) := I(l_{com,new}) \wedge \bigwedge_{1 \le i \le n}(stat_i == Id_{loc}(l_{b,i}))$
{$l_{b,i}$ is the i-th element of $\vec{l}_b$}

図 7 Duplication of States

DuplicateTransition
Inputs $\mathscr{A}', B_1 = (\vec{l}'_b, D_1), \vec{t}_b, l_{com,new}$

$\{\vec{t}_b = (\vec{l}_{prev}, a, g, r, \vec{l}_b)\}$
$a_{com} := newChannel()$ {generate a new channel}
$t_{com,new} := (l_{prev,com}, a_{com}, \emptyset, \emptyset, l_{com,new})$
{generate a transition to the duplicated location $l_{com,new}$}
$T_{com} := T_{com} \cup \{t_{com,new}\}$
$\vec{t}_{new} := (\vec{l}_{prev}, a_{com}, g, r, \vec{l}_b)$ {duplicate the transition $\vec{t}_b$}
$T := T \cup \{\vec{t}_{new}\}$
$Synchronize(t_{com,new}, \vec{t}_{new})$

foreach $\vec{t}_{succ} = (\vec{l}_1, a, g, r, \vec{l}_2) \in T$ such that $\vec{l}_1 = \vec{l}_b$ **do**
$Succ := Reach(\mathscr{A}, (\vec{l}_b, D_1), \vec{t}_{succ})$
{$Reach$ returns the reachable state set from $(\vec{l}_b, D_1)$}
if $Succ \neq \emptyset$ **then**
$a_{com} := newChannel()$ {generate a new channel}
$\vec{t}_{new} := (\vec{l}_b, a_{com}, g, r, \vec{l}_2)$
{duplicate the transition $\vec{t}_{succ}$}
$T := T \cup \{\vec{t}_{new}\}$
if A location corresponding to the state set $Succ$ is already generated **then**
$t_{com,new} := (l_{com,new}, a_{com}, \emptyset, \emptyset, l_{com,succ})$
{$l_{com,succ}$ is the location corresponding to $Succ$}
else
$t_{com,new} := (l_{com,new}, a_{com}, \emptyset, \emptyset, l_{com,other})$
end if
$T_{com} := T_{com} \cup \{t_{com,new}\}$
$Synchronize(t_{com,new}, \vec{t}_{new})$
$g_{new} := g \cup \{stat_{com} \neq Id_{loc}(l_{com,new})\}$
$setGuard(\vec{t}_{succ}, g_{new})$
{set the guard condition g_{new} into $\vec{t}_{succ}$}
end if
end for

図 8 Duplication of Transitions

たは同期する複数の遷移を仮想的に $\vec{l}'$ というように表記する．

状態の複製では 2.5 節の図 3 の状態集合 B_1 を複製する．提案手法は $(\vec{l}_b, D_1)$ を表現する合成ロケーション $l_{com,new}$ を生成し，$\mathscr{A}_{com}$ へ追加する．$\mathscr{A}_{com}$ に複製されたロケーション $l_{com,new}$ は複製の元となる $\vec{l} = (l_1, \ldots, l_n)$ について，$\bigwedge_{1 \le i \le n} stat_i == Id_{loc}(l_i)$ というインバリアントを付加し，他のプロセスの状態

RemoveTransition
Inputs $\mathscr{A}', B_1 = (\vec{l}'_b, D_1), \vec{t}_b = (\vec{l}_{prev}, a, g, r, \vec{l}_b), l_{com,new}$

$\{\vec{t}_b$: a transition to $\vec{l}_b\}$
if $l_{com,prev} \neq l_{other}$ **then**
foreach $t_{b,i} = (l_{prev,i}, a_i, g_i, r_i, l_{b,i})$ **such that** $t_{b,i}$ is an element of $\vec{t}_b$ and also a transition of $\mathscr{A}_i$ **do**
$g_i := g_i \cup \{stat_{com} \neq Id_{loc}(l_{com,new})\}$
end for
else
$I(\ l_{com,b}\ ) := I(\ l_{com,b}\ ) \wedge \bigwedge_{t_{b,i}\ is\ the\ element\ of\ \vec{t}_b} tr_i \neq Id_{tr}(t_{b,i})$
end if

図 9 Removal of Transitions

が $\vec{l}$ のときのみ l_{com} へ到達するようにしている．これは，Broadcast Channel で同期させる際に，意図しないプロセスとの同期を防ぐためである．状態の複製は図 7 のアルゴリズムによって行われる．

遷移の複製 (図 8) では複製した状態への遷移と複製した状態からの遷移をそれぞれ複製する．提案手法ではまず $l_{com,prev}$ から $l_{com,new}$ への遷移の複製 $t_{com,new}$ を生成し，$\vec{l}_{prev}$ から $\vec{l}_b$ への各遷移と同期させる (図 8 : 2–9 行目)．アルゴリズム中の関数 $Synchronize$ は $\mathscr{A}_{com}$ 上の遷移と $\mathscr{A}$ 上の遷移を同期させる関数である．遷移の同期については 3.2 節で示した方法で行う．さらに $\mathscr{A}$ において $(\vec{l}_b, D_1)$ からアクション遷移，時間遷移を一回ずつ行うことによって到達可能な状態を求める．そして $(\vec{l}_b, D_1)$ から到達可能な各状態 $(\vec{l}_{succ}, D)$ について $\mathscr{A}_{com}$ 上で対応する状態 $l_{com,succ}$ が存在するならば $l_{com,new}$ から $l_{com,succ}$ への遷移を複製する．対応する状態 $l_{com,succ}$ が存在しないならば $l_{com,new}$ から l_{other} への遷移として複製する．なお，これらの遷移は $\vec{l}_b$ から $\vec{l}_{succ}$ への各遷移と同期させる (図 8 : 10–32 行目)．また，28–29 行目では，$\mathscr{A}_{com}$ が $l_{com,new}$ である状態で複製の素となる遷移が実行されないように，遷移のガード制約を操作している．

遷移の削除 (図 9) では，$\vec{l}_{prev}$ から $\vec{l}_b$ への遷移を削除する．提案手法では，$\vec{l}_{prev}$ から $\vec{l}_b$ への各遷移に $stat_{com} \neq Id_{loc}(l_{com,prev})$ とおいうガード制約を追加し，$\mathscr{A}_{com}$ が $l_{com,prev}$ に滞在する間は $\vec{l}_{prev}$ から $\vec{l}_b$ への遷移を実行不可能にする．このとき，$l_{com,prev}$ が l_{other} である場合は，このような制約を加えると $\vec{l}_{prev}$ から $\vec{l}_b$ への遷移以外の遷移も削除されてしまう．このような場合は $l_{com,b}$ のインバリアントで，直前に実行する遷移を制限する制約を記述することで遷移の削除を実現している．$\vec{l}_{prev}$ から $\vec{l}_b$ への遷移を $\vec{t}_b$ とすると，具体的には $l_{com,b}$ のインバリアントに対して，

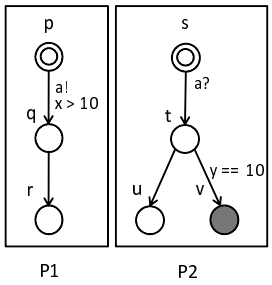

図 10 ネットワーク時間オートマトンの例

$\vec{t}_{prev}$ の各要素 $t_{b,i}$ について，$tr_i \neq Id_{tr}(t_{b,i})$ という制約を追加する．

このように偽反例を検出するたびにインクリメンタルに $\mathscr{A}_{com}$ が構成される．また，検出された偽反例がこれまでに検出した偽反例と独立であれば，偽反例を除去するためのパスがそれまでの $\mathscr{A}_{com}$ とは独立に生成されていく．

3.4.6 考　　察

提案手法では，ネットワーク時間オートマトン $\mathscr{A}$ に対して部分的な並列合成モデル $\mathscr{A}_{com}$ を用いて $\mathscr{A}$ の動作を制御している．$\mathscr{A}_{com}$ で制御するのは，クロック変数を持つ本来のモデルでは実行不可能となるような動作であり，本来制限すべきでない動作を制限することはない．したがって $\mathscr{A}$ に $\mathscr{A}_{com}$ を加えたモデルは $\mathscr{A}$ と等価であると言える．

また，抽象化洗練の各ループにおいて，状態の複製によって複製された状態 $l_{com,new}$ が $\mathscr{A}_{com}$ に追加されていくが，このとき，遷移の複製によって $l_{com,new}$ から到達可能な遷移はすべて複製しており，前回のループにおける $\mathscr{A}_{com}$ との整合性が保たれるようにしている．

3.5 洗練手法適用例

提案手法を用いて偽反例を取り除く例を示す．

図 10 はネットワーク時間オートマトンの例である．図 10 の例では，プロセス P2 のロケーション v を異常状態とし，異常状態への到達可能性を検証する．図 10 の例を初期状態から実行した場合，ロケーション (q, t) へ到達した時点で，$x == y \wedge x > 10$ が成り立っている．これより $y > 10$ も成り立つため，P2 の v へは到達不可能である．

このモデルに対して，部分合成モデル $\mathscr{A}_{com}$ を追加し，初期抽象化 (すべてのクロック変数を除去) を適用したモデルを図 11 に示す．$\mathscr{A}_{com}$ は初期状態 p_s と $other$ から構成されている．各プロセスの初期ロケーションからの遷移はすべて $\mathscr{A}_{com}$ の遷移 $(p\_s \rightarrow other)$ と同期させている．なお，遷移 $(p \rightarrow q)$，$(s \rightarrow t)$ は半

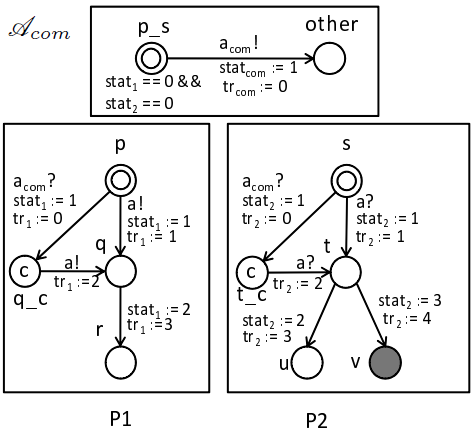

図 11 部分合成モデルを追加したモデル

アクションによる遷移であることから，$\mathscr{A}_{com}$ と同期させる際に Committed Location を介している．また，図 11 では $\mathscr{A}_{com}$ が p_s である状態のまま遷移 $(p \rightarrow q)$，$(s \rightarrow t)$ を実行することは，$\mathscr{A}_{com}$ のロケーション p_s のインバリアント $(stat_1 == 0)\&\&(stat_2 == 0)$ によって制限されている ($Id_{loc}(p) = 0$，$Id_{loc}(s) = 0$)．

図 11 のモデルに対してモデル検査を適用すると，$(p\_s, (p, s)) \overset{a_{com}}{\Rightarrow} (other, (q\_c, t\_c)) \overset{a}{\Rightarrow} (other, (q, t)) \Rightarrow (other, (q, v))$
という反例が検出されるが，これは偽反例である．この偽反例に対するモデルの洗練では，状態集合 $((q, t), D_1$ の複製を行う．なお D_1 は $x == y \wedge x > 10$ を満たすような状態の集合である．提案手法では $((q, t), D_1)$ を意味する状態 q_t を $\mathscr{A}_{com}$ へ生成し，遷移 $(p \rightarrow q\_c)$，$(s \rightarrow t\_c)$ と遷移 $(p\_s \rightarrow q\_t)$ 同期させ，P2 の遷移 $(t \rightarrow v)$ に $(stat_{com}! = 2)$ というガード制約を与えることで，偽反例をモデルから除去している．提案手法による洗練を適用したモデルを図 12 に示す．

4. 実　　験

本章では，提案手法による抽象化洗練手法を実装し，例題に適用した結果を示す．実験では実行時間，メモリ消費量という点について既存手法との比較を行う．なおモデル検査では UPPAAL のモデル検査モジュール verifyta を利用している．実験環境は以下の通りである．

OS	:	Fedora 7
CPU	:	AMD Athlon(tm) 64 Processor 3400+ 2.2GHz
Memory	:	930MB
UPPAAL	:	version 4.0.6

提案手法を適用する例題として，Fischer の相互排除

表 1 Fischer の相互排除プロトコルに対する実験結果

		既存手法			提案手法		
Proc	clock	time	mem	loop	time	mem	loop
2	2	0.74s	2.82MB	5	2.06s	19.5MB	3
3	3	2.44s	2.82MB	13	15.1s	36.9MB	15
4	4	10.9s	21.6MB	25	36.6s	37.6MB	28
5	5	69.4s	48.1MB	41	102s	39.5MB	45
6	6	662s	95.0MB	61	360s	45.0MB	66
7	7	12571s	293MB	85	1329s	61.0MB	91
8	8	N/A	N/A	N/A	4723s	129MB	120

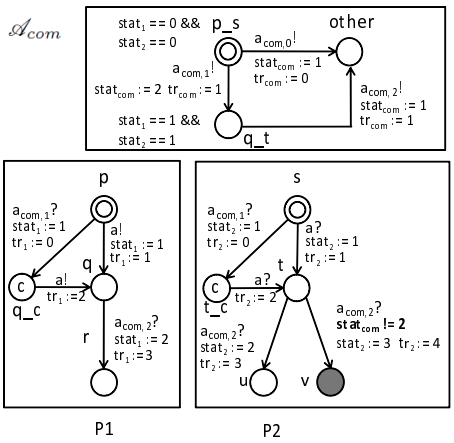

図 12 洗練後のモデル

プロトコル[7]，を選択し実験を行った．実験結果を表 1 に示す．Proc はモデルを構成するプロセス数，Clock はモデル全体のクロック変数の個数である．既存手法の列は事前にモデル全体を並列合成し，抽象化を適用した結果を示し，提案手法の列は提案手法による実験結果を示している．time は CEGAR ループ全体の実行時間，mem はモデル検査で消費した最大のメモリ量，loop は CEGAR ループの繰り返し回数である．既存手法の 8 プロセスに対する実験では，並列合成によりロケーション数が 4^8 個となり verifyta の仕様により実行不可能であった．

表 1 の結果より，5 プロセス以上では提案手法がメモリ消費量の点で優れていることがわかる．特に 7 プロセスではメモリ消費量を約 80 パーセント削減でき，既存手法では適用不可能であった 8 プロセスのモデルでも適用できるなど，提案手法が有効であることを示している．さらに，実行時間についても 7 プロセスでは約 90 パーセント削減しており，実行時間の点でも提案手法が優れた結果を示すことができた．

提案手法と既存手法で CEGAR ループの回数が異なっているのは，提案手法と既存手法では入力となるモデルの表現方法が異なっている (並列プロセスモデルと並列合成後のモデル) ため，verifyta による状態探索の順序に違いが生じ，発生した反例が異なったためであると考えられる．CEGAR ループによる抽象モデルの洗練は発生する反例に依存するため，このような差が生じたものと考えられる．

なお，文献 6) では，本例題に対して純粋に UPPAAL のみを用いて検証を行った結果を示しており，各プロセスの対称性を利用した Symmetry Reduction により，メモリ消費量はプロセス数に対して線形的な増加に抑えられている．よって本例題のようにプロセス間に対称性があるような例題に対しては提案手法による抽象化よりも Symmetry Reduction が有効であると考えられる．今後の課題としては，プロセス間に対称性が存在せず Symmetry Reduction を利用できない例題に対して提案手法を適用し，提案手法による抽象化の有効性を評価することが挙げられる．また，このような対称性を利用し，抽象モデルの洗練を効率的に行う方法の考案も課題として挙げられる．

5. おわりに

本稿では，著者らがこれまでに提案してきた時間オートマトンに対する抽象化洗練手法に関する問題点の改善を行った．既存の抽象化手法では, 複数のプロセスが並列に動作するようなモデルを検査する場合，事前に複数のプロセスを並列合成する必要があり，抽象化効率の低下の原因となっていた．本稿では，このような問題を解決するため，そのような合成を部分的にとどめるような手法を提案した．また提案手法を簡単な例題に対して適用し，メモリ消費量，実行時間の両面において既存手法を改善できていることを示した．

今後の課題としては，アルゴリズムを並列化し，抽象化全体の実行時間を縮小することが挙げられる．また，より規模の大きい例題に適用し，提案手法の有効性を評価したいと考えている．

参 考 文 献

1) E. Clarke, O. Grumberg, S. Jha, Y. Lu, and V. Helmut: “Counterexample-guided Abstraction Refinement,” Computer Aided Verification:12th International Conference, vol.1855, pp.154-169, 2000.
2) E. Clarke, A, Gupta, J. Kukula, and O. Strichman: “SAT based Abstraction-Refinement using ILP and Machine Learning Techniques,” Computer Aided Verification:14th International Conference, vol.2404, pp.695-709, 2002.
3) R. Alur: “Techniques for Automatic Verification of Real-Time Systems,” PhD thesis, Stanford University, 1991.
4) R. Alur, C.Courcoubetis, and D. L. Dill: “Model-checking for real-time systems,” In Proceedings of the 5th Annual Symposium on Logic in Computer Science, pp.414-425. IEEE Computer Society Press, 1990.
5) J. Bengtsson, and W.Yi: “Timed Automata: Semantics, Algorithms and Tools,” Lectures on Concurrency and Petri Nets, vol.3098, pp.87-124, 2004.
6) G.Behrmann, A.David, K. G. Larsen, J. Håkansson, P. Pettersson, W. Yi and M. Hendriks: “UPPAAL 4.0,” In Proc. of the 3rd Int. Conf. on the Quantitative Evaluation of Systems (QEST), pp125-126, IEEE Computer Society, 2006,
7) G. Behrmann, A. David, and K G. Larsen: “A Tutorial on UPPAAL,” In Proc. of the 4th Int. School on Formal Methods for the Design of Computer, Communication, and Software Systems, vol.3185, pp.200-236, 2004
8) J. Håkansson, and P. Pettersson: “Partial Order Reduction for Verification of Real-Time Components,” In Proc. of the 5th Int. Conf of Formal Modelling and Analysis of Timed Systems, vol.4763, pp211-226, 2007.
9) 長岡 武志，岡野浩三，楠本真二: “UPPAAL 拡張時間オートマトンの反例に基づく抽象化改良ループによるモデル抽象化手法,” 電子情報通信学会技術研究報告，vol.107，No.176，pp.77-82，2007
10) T. Nagaoka, K. Okano, and S. Kusumoto: “Abstraction of Timed Automata Based on Counterexample-Guided Abstraction Refinement Loop,” IEICE Technical Report, vol.107, No.505, pp.103-108，2008.