グリッド計算機を利用した並列仕様検証に関する研究

Parallel Verification of Specification with Grid-Computing

大滝 大輔 † 安藤 崇央 † 米崎 直樹 †
Daisuke OHTAKI Takahiro ANDO Naoki YONEZAKI
† 東京工業大学大学院情報理工学研究科計算工学専攻
Dept. of Computer Science, Graduate School of Information Science and Engineering,
Tokyo Institute of Technology
{ohtaki, ando, yonezaki}@fmx.cs.titech.ac.jp

リアクティブシステムのあらゆる状況に対する動作可能性に関する仕様レベルでの検証は，実現されたシステムの検証よりも効果が高いと言われている．しかし，1 台の計算機上に実装した既存の証明器では，現実規模の動作仕様の性質判定に限界がある．本稿では，動作仕様の実現可能性の必要条件である充足可能性，および段階的充足可能性の判定手続きを，並列計算用の代表的ライブラリである MPI を用いて複数台の計算機上に実装する手法を提案する．さらに，提案した並列化手法を用いて実際に Java 言語により検証器をグリッドシステム上で実装し，本手法が台数効果を十分発揮できることを実験により確かめたことを報告する．

1 はじめに

1.1 背景と目的

リアクティブシステムとは，エレベータやオペレーティングシステムのような外部からの要求に対して，定められた仕様を満たすように応答し続けるシステムの事である．このようなシステムは，外部からの要求が複雑に絡み合うため，あらゆる状況に対しても正しく応答することが可能という，動作仕様の実現可能性は非常に失われやすい．加えて，その性質判定にはモデル検査に比べて膨大な時間がかかる．しかし，実現可能な仕様が満たす必要条件である充足可能性や段階的充足可能性などについて，検証を行うことで，仕様が実現不可能であったときに，その原因を少ない時間で解析することが可能である．また，この性質の検証方法として，タブロー法に基づいた手続きが提案されている [5][6]．タブロー法では，動作仕様からタブローグラフと呼ばれるグラフを構成し，このグラフを解析する事で仕様の性質について検証を行う．ここで，素朴にタブローグラフを構成した場合，システムの状態が数多く生成されることが，主な問題となる．一般に素朴な方法を適用すれば，手に負えない状態数を処理する事になるが，BDD 表現やプレステート表現 [6] を駆使することにより，処理時間の大幅な削減を図る事が可能である．しかし，なお状態数の爆発が実際的応用の幅を狭めている．

一方近年では，コンピュータ端末の高速化，低価格化にともない，高価格なスーパーコンピュータを用意するより，パーソナルコンピュータを複数台繋げる事で安価でそれに近いスペックを実現するグリッドコンピューティングが発達してきている．

そこで，もしタブロー法等の仕様検証が，グリッドを使い並列に行うことが可能ならば，検証を高速化し，より大きな仕様に対し行う事が可能であると考えられる．そのため本研究では，タブロー法に対する並列化手法を定め，グリッド計算機上で，その手法を用いたエンジンを実装する事を目的とする．

1.2 本稿の概要

本稿では，実際にタブロー法に対する並列化手法を提案し，その手法を用いた検証エンジンを，実際に並列計算用の代表的なライブラリである Message Passing Interface [10][12] を用いて実装した．また，グリッド計算機上で実験を行い，実際に高速化されることを確認し，本手法が台数効果を十分に発揮できる事を示す．加えて，メモリに対する効果に関してその結果から言及する．まず 2 節，3 節で動作仕様に関する各種性質判定の関連や，判定方法について述べ，4 節で並列化アルゴリズムの説明を行う．5 節では，実際にグリッド計算機上でおこなった実験結果の，評価を行い，6 節で今回のまとめと，今後の目標について述べる．

2 準備

本節ではリアクティブシステムと，その動作仕様に関する諸定義を与える．基本的にこれらの定義は [4] に準じたものを用いる．

2.1 リアクティブシステム

定義 2.1 ($A^+, A^\omega, A^\dagger$)
A を有限集合とするとき，A 上の空列を含まないすべての有限列の集合を A^+ と表し，すべての可算無限列の集合を A^ω と表す．また，これらの集合の和集合 $A^+ \cup A^\omega$ を $A^\dagger$ と表す．

列 $\tilde{a} \in A^\dagger$ の長さを $|\tilde{a}|$ と表し，$\tilde{a} \in A^+$ ならば $|\tilde{a}|$ は列に含まれる A の要素の個数，$\tilde{a} \in A^\omega$ ならば $|\tilde{a}| = \omega$ と定める．また，$\tilde{a}$ の i $(0 \le i < |\tilde{a}|)$ 番目の要素を a_i で表す．2 つの列 $\tilde{a}, \tilde{a}'$ の連結を $\tilde{a} \cdot \tilde{a}'$ で表す．ただし $\tilde{a} \in A^\omega$ のときは，$\tilde{a} \cdot \tilde{a}' = \tilde{a}$ とする．

定義 2.2 (リアクティブシステム)
リアクティブシステムは，三つ組み $RS = \langle \mathbb{R}, \mathbb{S}, \mathbf{r} \rangle$ である．三つ組みの要素はそれぞれ以下の通りである．

- $\mathbb{R}$ は**要求イベント**の集合である．要求イベントとはシステム外部の環境から入力されるイベントのことである．また，$(2^{\mathbb{R}})^\dagger$ の各要素を**要求イベント列**と呼ぶ．
- $\mathbb{S}$ は**応答イベント**の集合である．応答イベントとはシステムが生起させるイベントのことである．また，$(2^{\mathbb{S}})^\dagger$ の各要素を**応答イベント列**と呼ぶ．
- $\mathbf{r} : (2^{\mathbb{R}})^+ \to 2^{\mathbb{S}}$ は**リアクション関数**である．リアクション関数はその時点までの有限の要求イベント列の入力に対して，システムがどのような応答イベントを生起させるかを決定する関数である．

定義 2.3 (ふるまい)
イベントの集合の列を**ふるまい**と呼ぶ．要求イベント列 $\tilde{r} = r_0 r_1 r_2 \cdots \in (2^{\mathbb{R}})^\dagger$ に対するリアクティブシステム $RS = \langle \mathbb{R}, \mathbb{S}, \mathbf{r} \rangle$ のふるまい $\mathrm{behavior}(\mathbf{r}, \tilde{r}) \in (2^{\mathbb{R} \cup \mathbb{S}})^\dagger$ を以下のように定義する．

$$\mathrm{behavior}(\mathbf{r}, \tilde{r}) = (r_0 \cup s_0)(r_1 \cup s_1)(r_2 \cup s_2) \cdots$$

ただし $s_i = \mathbf{r}(r_0 r_1 r_2 \cdots r_i)$, $(0 \le i)$ とする．また，$\tilde{r} \in (2^{\mathbb{R}})^\dagger$, $\tilde{s} \in (2^{\mathbb{S}})^\dagger$ とするとき $\langle \tilde{r}, \tilde{s} \rangle$ は，i 番目の状態が $(r_i \cup s_i)$ となるふるまいを表すものとする．

2.2 動作仕様

リアクティブシステムの動作仕様は，システムの可能なふるまいの集合を定めるものである．本稿では，命題線形時間論理 **PLTL**(Propositional Linear Temporal Logic) により記述された動作仕様に関する検証を扱う．PLTL は様相演算子として，いわゆる「**weak until** 演算子 [$\cdot$]」[15] を含む古典命題線形時間論理である．

定義 2.4 (論理式)
$\mathcal{P}$ を命題変数の集合とするとき，$\mathcal{P}$ 上の式は次のように構成される．命題変数 $p \in \mathcal{P}$ は式である．f, g が式であるとき，$\neg f$， $f \wedge g$， $[g]f$ は式である．また，$f \vee g$， $f \to g$， $\langle g \rangle f$， $\Box f$， $\Diamond f$，$\top$ は，それぞれ $\neg(\neg f \wedge \neg g)$， $\neg f \wedge g$， $\neg[g]\neg f$， $[\bot]f$， $\langle \bot \rangle f$， $\neg\bot$ の略記とする．ここで $\bot = p \wedge \neg p$, $p \in \mathcal{P}$ である．

定義 2.5 (動作仕様)
リアクティブシステムの動作仕様は，三つ組み $Spec = \langle \mathcal{R}, \mathcal{S}, \varphi \rangle$ である．ただし

- $\mathcal{R}$ は，要求イベントに対する命題変数の有限集合である．その命題の成立は対応する要求イベントの生起を表す．$\mathcal{R}$ の要素を**要求変数**と呼ぶ．
- $\mathcal{S}$ は，応答イベントに対する命題変数の有限集合である．その命題の成立は対応する応答イベントの生起を表す．$\mathcal{S}$ の要素を**応答変数**と呼ぶ．
- φ は，$\mathcal{R} \cup \mathcal{S}$ に含まれる命題変数からなる PLTL の式である．

定義 2.6 (意味)
$\mathbb{P}$ をイベントの集合，$\mathcal{P}$ を $\mathbb{P}$ に対する命題変数の集合とする．ふるまい $\sigma \in (2^{\mathbb{P}})^\dagger$ の i 番目の状態が $\mathcal{P}$ 上の式 f を満たすことを $\langle \sigma, i \rangle \models f$ と表し，以下のように帰納的に定義する．

- $\langle \sigma, i \rangle \models p \iff p' \in \sigma_i$ ($p' \in \mathbb{P}$ は $p \in \mathcal{P}$ に対応するイベント)
- $\langle \sigma, i \rangle \models \neg f \iff \langle \sigma, i \rangle \not\models f$
- $\langle \sigma, i \rangle \models f \wedge g \iff \langle \sigma, i \rangle \models f$ かつ $\langle \sigma, i \rangle \models g$
- $\langle \sigma, i \rangle \models [g]f \iff$
  $(\forall j \ge 0)\, \langle \sigma, i + j \rangle \models f$ または
  $(\exists j \ge 0) \Big(\langle \sigma, i + j \rangle \models g$ かつ
  $\forall k (0 \le k < j)\, \langle \sigma, i + k \rangle \models f \Big)$

以下，$\langle \sigma, 0 \rangle \models f$ を単に $\sigma \models f$ と表記する．

3 動作仕様が満たすべき性質と検証法

本稿では，リアクティブシステムの動作仕様が満たすべき性質のうち，以下に示す**充足可能性**と**段階的充足可能性**という 2 つの性質の判定手続きについて扱う．動作仕様が満たすべき性質として，充足可能性，段階的充足可能性のほかに，**実現可能性**，**強充足可能性**，**段階的強充足可能性**という性質が提案されている [5]．本稿では，実現可能性，強充足可能性，段階的強充足可能性については簡単な紹介にとどめる．

実現可能性 システム外部の環境がどのようにふるまっても，システムが動作仕様を満たしながら今までに観測した要求から応答を決定し続けることができるシステムが存在する．

充足可能性 無限要求イベント列と無限応答イベント列が存在し，それらのイベント列からなるふるまいが動作仕様を満たす．

段階的充足可能性 実行中のどの時点においても，充足可能性を失わないように任意の要求に対して動作仕様を満たしながら応答できるシステムが存在する．

強充足可能性 任意の無限要求イベント列に対して，動作仕様を満たす応答列が存在する．

段階的強充足可能性 実行中のどの時点においても，強充足可能性を失わないように任意の要求に対して動作仕様を満たしながら応答できるシステムが存在する．

それぞれの性質の包含関係は図 1 の様になる．段階的充足可能性は実現可能性の必要条件であるが，実用上は有限長の任意の入力列に対して動作仕様を満たしながら応答するシステムが存在することが保証されれば十分である場合が多い．さらに，実現可能性判定の計算コストが段階的充足可能性と比べ大きいため，段階的充足可能性を判定する意義は高い．以下の節では，本稿で扱う充足可能性，段階的充足可能性の形式的な定義とそれぞれの判定手続き [4][1] を紹介する．

3.1 充足可能性

動作仕様が**充足可能**であるとは,「無限要求イベント列と無限応答イベント列が存在し，それらのイベント列からなるふるまいが動作仕様を満たす」ことを表す．以下に充足可能性の形式的な定義を与える．

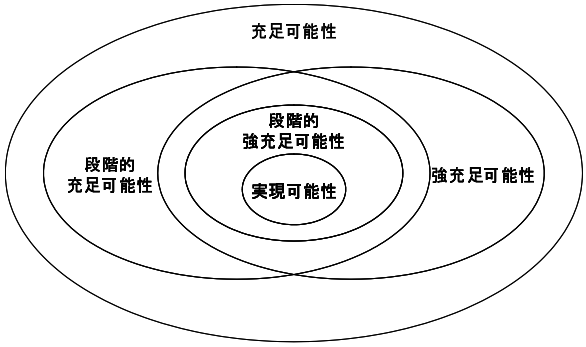

図 1: 動作仕様の満たすべき性質の包含関係

定義 3.1 (充足可能性)
リアクティブシステムの動作仕様 $Spec = \langle \mathcal{R}, \mathcal{S}, \varphi \rangle$ が次の条件を満たすとき，その動作仕様は**充足可能**であるという．

$$\exists \tilde{r} \in (2^{\mathbb{R}})^{\omega}, \exists \tilde{s} \in (2^{\mathbb{S}})^{\omega} \left(\langle \tilde{r}, \tilde{s} \rangle \vDash \varphi \right)$$

ここで，$\mathbb{R}$，$\mathbb{S}$ はそれぞれ，$\mathcal{R}$，$\mathcal{S}$ に対応するイベントの集合である．

3.2 段階的充足可能性

動作仕様が**段階的充足可能**であるとは,「実行中のどの時点においても，充足可能性を失わないように任意の要求に対して動作仕様を満たしながら応答できるシステムが存在する」ことを表す．リアクティブシステムが実際に動作する状況を考える．実行中のある段階において，ある要求イベントに対して動作仕様を満たす応答イベントが存在しなかった場合，過去にさかのぼって全く同じ要求イベント列に対する別の応答イベント列を選び直すことができないことは自明である．すなわち，リアクティブシステムの動作仕様が実現可能であるためには,「すべての無限要求イベント列に対して，動作仕様を満たす応答列が存在する」という強充足可能性を満たすだけでは不十分である．未来を予測することなく，実行中の各段階において動作仕様を満たす様に応答する方法が存在する必要がある．この条件を満たす性質が段階的充足可能性である．以下に段階的充足可能性の形式的定義を与える．

定義 3.2 (段階的充足可能性)
リアクティブシステムの動作仕様 $Spec = \langle \mathcal{R}, \mathcal{S}, \varphi \rangle$

が次の条件を満たすとき，その動作仕様は**段階的充足可能**であるという．

$$\exists \mathbf{r}\left(\forall \tilde{r} \in (2^{\mathbb{R}})^{+}, \exists \tilde{r}' \in (2^{\mathbb{R}})^{\omega}, \exists \tilde{s}' \in (2^{\mathbb{S}})^{\omega} \left(\text{behavior}(\mathbf{r}, \tilde{r}) \cdot \langle \tilde{r}', \tilde{s}' \rangle \vDash \varphi\right)\right)$$

ここで，$\mathbf{r}$ はリアクション関数，$\mathbb{R}$， $\mathbb{S}$ はそれぞれ，$\mathcal{R}$， $\mathcal{S}$ に対応するイベントの集合である．

3.3 タブロー法

前節までに与えた動作仕様の充足可能性を判定する方法として，タブロー法に基づく判定法が提案されている [4]．また，段階的充足可能性については [1] の手続きを基に，タブロー法に適応したものを用いる．それぞれのタブロー法により構成されるタブローグラフの特徴から動作仕様の充足可能性，および段階的充足可能性の性質を判定することができる．以下にタブロー法に関する諸定義とタブローグラフの構成方法を与える．タブローグラフの構成手続きは，式の分解手続きとグラフの構成手続きに分けられる．

定義 3.3 (時間式)
f, g を式としたとき，$[g]f, \neg[g]f$ の形をした式を**時間式**と呼ぶ．S を式の集合とするとき S に含まれる時間式の集合を $\mathit{Temp}(S)$ と表す．

定義 3.4 (イベンチャリティ式)
f, g を式としたとき，$\neg[g]f$ の形をした式を**イベンチャリティ式**と呼ぶ．

定義 3.5 (ステートグラフ, ステート)
ステートグラフは三つ組み $\langle V_S, v_0, E \rangle$ で表す．V_S は**ステート**の集合，$v_0 \in V_S$ は**開始ステート**，$E \subseteq V_S \times V_S$ は**エッジ**の集合である．ステートは式の集合である．

ステートに含まれる命題式の肯定的出現が，対応するイベントの生起を表す．また，逆に命題式の否定的出現は対応するイベントが起こらないことを表す．

定義 3.6 (自己充足)
次の 2 つの条件を満たす部分ステートグラフ G を**自己充足**であると呼ぶ．

- G 中に少なくとも 1 つのエッジを持つ．
- G に含まれるステート s に含まれるすべてのイベンチャリティ式 $\neg[g]f$ について，G 内に s から到達可能な $\neg f, \neg g$ を含むステートが存在する．

手続き 3.1 (式の分解手続き *decomp*)
式集合 S を入力とし，分解した結果の式集合の集合 Σ を出力する．Σ は次の手順で構成する．また以下では式集合 S に分解手続きを適用した結果の集合を $\mathit{decomp}(S)$ と表す．

1. (初期化) $\Sigma := \{S\}$ とする．
2. (分解) 任意の式集合 $S_i \in \Sigma$ 中の任意の式 f_{ij}(リテラルは除く) について，f_{ij} の形に応じて以下の (a)〜(e) のいずれかを適用する．どの f_{ij} に適用しても Σ が変化しなくなるまで，これを繰り返す．
   (a) f_{ij} が $\neg\neg f$ という形の式ならば，S_i を以下の集合で置き換える．
   $$(S_i - \{f_{ij}\}) \cup \{f\}$$
   (b) f_{ij} が $f_1 \wedge f_2$ という形の式ならば，S_i を以下の集合で置き換える．
   $$(S_i - \{f_{ij}\}) \cup \{f_1, f_2\}$$
   (c) f_{ij} が $\neg(f_1 \wedge f_2)$ という形の式ならば，S_i を以下の 2 つの集合で置き換える．
   $$(S_i - \{f_{ij}\}) \cup \{\neg f_1\},\ (S_i - \{f_{ij}\}) \cup \{\neg f_2\}$$
   (d) f_{ij} が $[f_2]f_1$ という形の式ならば，S_i を以下の 2 つの集合で置き換える．
   $$(S_i - \{f_{ij}\}) \cup \{f_2\},$$
   $$(S_i - \{f_{ij}\}) \cup \{f_1, \neg f_2, [f_2]f_1\}$$
   (e) f_{ij} が $\neg[f_2]f_1$ という形の式ならば，S_i を以下の 2 つの集合で置き換える．
   $$(S_i - \{f_{ij}\}) \cup \{\neg f_1, \neg f_2\},$$
   $$(S_i - \{f_{ij}\}) \cup \{f_1, \neg f_2, \neg[f_2]f_1)\}$$
3. 式 f と $\neg f$ を同時に含む Σ の要素をすべて Σ から取り除く．

式の分解手続きは，それぞれの式の意味に沿って分解を行ってゆく．その過程で矛盾する式が含まれる式集合が現れた場合には，その式集合を削除する．また，次のステートにおいても真となるべき時間式

はそのまま残す．例えば式 $[g]f$ の意味は「式 g が真となるまで，常に式 f が真」であり，この式に分解手続きを適用すると，式 $[g]f$ は 2 つの式集合 $\{g\}$(式 g が真となる場合) と，$\{f, \neg g, [g]f\}$ (未だ式 g が真でない場合，式 f は真であり，式 $[g]f$ は次の時間にも引き継がれる) に分解される．

以上に述べた式の分解手続き *decomp* を以下のように繰り返し用いることでステートグラフを構成することができる．

手続き 3.2 (ステートグラフ構成 *constSGraph*)
式 φ を入力とし，ステートグラフ $\langle V_S, v_0, E\rangle$ を出力する．

```
1:  constSGraph(φ : formula) {
2:      v0 := {φ};
3:      VS := {v0};
4:      E := ∅;
5:      V'S := ∅;
6:      foreach (s ∈ VS − V'S) {
7:          if (s = v0)
8:              S := decomp(s);
9:          else
10:             S := decomp(Temp(s));
11:         VS := VS ∪ S;
12:         foreach (s' ∈ S) {
13:             E := E ∪ {⟨s, s'⟩};
14:         }
15:         V'S := V'S ∪ {s};
16:     }
17: }
```

5–6 行目: V'_S は分解済みのステートの集合を表す．s にはまだ分解されていないステートが入る．
10 行目: s が開始ステートでない場合，時間式のみを抽出した式集合を分解する．開始ステートの式は，それ自身が最初のステートが満たすべき制約を意味するため時間式を抽出する時間遷移規則を適用する必要がない．

式 $\Box(r \rightarrow \Diamond s)$ に対して，ステートグラフ構成手続きによってステートグラフを構成した例を図 2 に示す．ここではステートを角の丸い四角形で表している．網掛けのかかったステートは，開始ステートを表している．各ステートに付記した数字は，そのステートのラベルである．

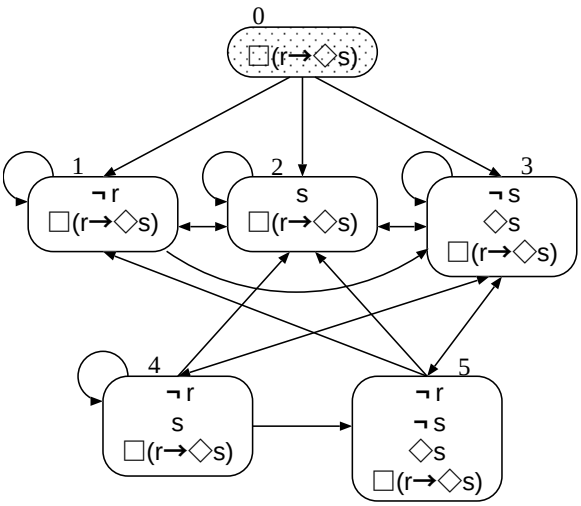

図 2: ステートグラフ

3.4 充足可能性判定

3.3 節で与えたステートグラフ構成手続き 3.2 を用いて，リアクティブシステム動作仕様の充足可能性は以下の手続きで判定される．

手続き 3.3 (充足可能性判定)
リアクティブシステムの動作仕様 $Spec = \langle \mathcal{R}, \mathcal{S}, \varphi\rangle$ を入力とし，*Spec* が充足可能であれば真を，そうでなければ偽を出力する．

1. G := *constSGraph*(φ) により，ステートグラフ G を構成する．
2. G を極大強連結成分に分割する．ここで自己充足となる部分グラフが存在するならば真を，そうでなければ偽を出力する．

動作仕様を満たす要求イベントと応答イベントの組の列が一つでも存在すれば，その動作仕様は充足可能である．ステートグラフに含まれるすべてのステートは，その開始ステートから到達可能であることを考えると，自己充足となる強連結成分がステートグラフ中に含まれることは無限の要求イベント列とこれに対応した動作仕様を満たす応答イベント列の組が少なくとも一つ存在していることを意味する．

3.5 段階的充足可能性判定

次に動作仕様の段階的充足可能性判定手続きについて述べる．段階的充足可能性の判定手続きでは，手続き 3.2 によりステートグラフを構成し，そのグ

ラフからイベンチャリティ式が含まれ，かつそのイベンチャリティ式が満たされるステートに到達不可能なステートを削除する．さらに，リアクティブシステムの状態をステートの集合ととらえるために[1]，要求変数および応答変数によって決定化する必要がある．

以下では，ステートグラフの決定化について述べる [9][1]．準備としてまず 3 つの関数と，ステートグラフを命題変数の集合で決定化したグラフである**決定性ステートグラフ**を定義する．

定義 3.7 (atm, $\overline{\text{atm}}$, suc)
ステートグラフ $G = \langle V_S, v_0, E\rangle$ に対して以下を定義する．ただし $s \in V_S$ はステート，$\mathbf{m} \in 2^{V_S}$ はマクロステートの集合，p は命題変数とする．

- $\mathbf{atm}(s) \stackrel{\text{def}}{=} \{p \mid p \in s\}$
- $\overline{\mathbf{atm}}(s) \stackrel{\text{def}}{=} \{p \mid \neg p \in s\}$
- $\mathbf{suc}(\mathbf{m}) \stackrel{\text{def}}{=} \bigcup_{m \in \mathbf{m}} \{m' \mid \langle m, m'\rangle \in E\}$

定義 3.8 (決定性ステートグラフ)
$\langle V, v_0, E\rangle$ を式 φ のステートグラフとするとき，以下の条件を満たす三つ組み $\langle \mathcal{M}, \mathbf{m}_0, \mathcal{E}\rangle$ を式 φ と命題変数の集合 $\mathcal{P}$ の**決定性ステートグラフ**と呼ぶ．

- $\mathcal{M}$ はステート集合を要素に持つ，空集合を含まない集合．すなわち $\mathcal{M} \subseteq 2^V$
- $\mathbf{m}_0 = \{v_0\} \in \Updownarrow$
- $\mathcal{E}$ は $\langle \mathbf{m}, \mathbf{p}, \mathbf{m}'\rangle$ の集合．ただし，$\mathbf{m}, \mathbf{m}' \in \mathcal{M}$, $\mathbf{p} \subseteq \mathcal{P}$
- $\langle \mathbf{m}, \mathbf{p}, \mathbf{m}'\rangle \in \mathcal{E}$ ならば，$\mathbf{p} \cap \overline{\mathbf{atm}}(\mathbf{m}') = \emptyset$ かつ $(\mathcal{P} - \mathbf{p}) \cap \mathbf{atm}(\mathbf{m}') = \emptyset$
- $\langle \mathbf{s}, \mathbf{p}, \mathbf{s}'\rangle \in \mathcal{E}$ ならば，$\forall s' \in \mathbf{m}', \exists s \in \mathbf{m}\left(\langle s, s'\rangle \in E\right)$

ここまでの定義により，ステートグラフは以下の手続きによって決定化される．

手続き 3.4 (ステートグラフ決定化手続き *convDet*)
ステートグラフ $G = \langle V_S, v_0, E\rangle$ と命題変数の集合 $\mathcal{P}$ を入力とし，$\mathcal{P}$ の要素からなる列に関して，G を決定化した決定性ステートグラフ $\mathcal{G} = \langle \mathcal{M}, \mathbf{m}_0, \mathcal{E}\rangle$ を出力する．ここで，S をステートの集合とするとき

$$S \setminus \mathbf{p} \stackrel{\text{def}}{=} \{s \in S \mid \mathbf{p} \cap \overline{\mathbf{atm}}(s) = \emptyset,\ (\mathcal{P} - \mathbf{p}) \cap \mathbf{atm}(s) = \emptyset\}$$

とする．

```
1:  convDet(G : state graph,
              P : set of atomic propositions) {
2:    m0 := {v0};
3:    M := {m0};
4:    E := ∅;
5:    M' := ∅;
6:    foreach (m ∈ M − M') {
7:      foreach (p ∈ 2^P) {
8:        if ((suc(m) \ p) ≠ ∅) {
9:          M := M ∪ {suc(m)};
10:         E := E ∪ {⟨s, p, (suc(m) \ p)⟩};
11:       }
12:     }
13:     M' := M' ∪ {m};
14:   }
15: }
```

5–6 行目: $\mathcal{M}'$ は処理済みの $\mathcal{M}$ の要素の集合．最終的に，$\mathcal{M}$ に含まれるすべての要素についてちょうど一回ずつ，遷移先となるステート集合を作る操作を行う．

以下の図 3 に，図 2 のステートグラフを命題変数の集合 $\{r, s\}$ で決定化した決定性ステートグラフを示す．図中の各数字は，図 2 中の各ステートのラベルである．つまり図 3 においてラベルが 1, 2 であるマクロステートは，図 2 においてラベルが 1 のステートとラベルが 2 のステートの集合を表す．

ある要求イベントの組み合わせに対して動作仕様を満たす応答が存在しないマクロステート集合を**行き止まり**と呼ぶ．以下に行き止まりの定義を与える．

定義 3.9 (行き止まり)
$\langle \mathcal{V}, \mathbf{v}_0, \mathcal{E}\rangle$ を動作仕様 $\langle \mathcal{R}, \mathcal{S}, \varphi\rangle$ から構成された決定性ステートグラフとする．マクロステート集合 $\mathbf{m} \in \mathcal{M}$ が次の条件を満たすとき，$\mathbf{m}$ は**行き止まり**であるという．

$$\exists \mathbf{x} \in 2^{\mathcal{R}}, \forall \mathbf{y} \in 2^{S}, \forall \mathbf{m}' \in \mathcal{M}\left(\langle \mathbf{m}, \mathbf{x} \cup \mathbf{y}, \mathbf{m}'\rangle \notin \mathcal{E}\right)$$

以上により，動作仕様の段階的充足可能性は以下の手続きで判定される．

[1] [4] で提案された手続きでは，ステートの集合をリアクティブシステムの状態としてとらえられなかったため，段階的充足可能性の判定が正しく行えない式が存在した．

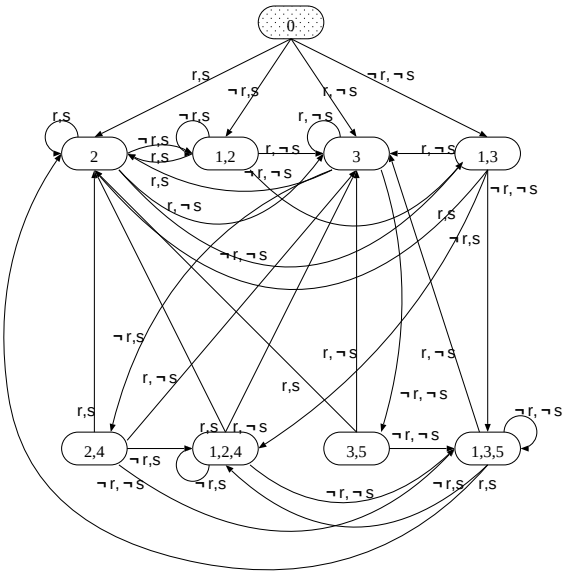

図 3: 決定性ステートグラフ

手続き 3.5 (段階的充足可能性判定)
リアクティブシステムの動作仕様 $Spec = \langle \mathcal{R}, \mathcal{S}, \varphi \rangle$ を入力とし，$Spec$ が段階的充足可能であれば真を，そうでなければ偽を出力する．

1. G := $constSGraph(\varphi)$ により，ステートグラフ G を構成する．
2. 以下のステップ (a)–(c) を G が変化しなくなるまで繰り返す．
   (a) イベンチャリティ式が含まれ，かつそのイベンチャリティ式が満たされるステートに到達不可能なステートを削除する．
   (b) 削除されたステートに入るエッジ，及び削除されたステートから出るエッジをすべて削除する．
   (c) G の開始ステートが削除された場合には，偽を出力する．
3. ステップ 2. により変更された G を手続き 3.4 により決定化し，決定性ステートグラフ $\mathcal{G}$:= $convDet(G, \mathcal{R} \cup \mathcal{S})$ を構成する．
4. 以下の操作を $\mathcal{G}$ が変化しなくなるまで繰り返す．
   (a) $\mathcal{G}$ 中の行き止まりのマクロステートを削除する．
   (b) 削除されたマクロステートに入るエッジ，及びそのマクロステートから出るエッジをすべて削除する．
5. $\mathcal{G}$ の開始マクロステートが削除された場合に偽を，そうでなければ真を出力する．

上記の手続きによって，最終的に得られた決定性ステートグラフに開始マクロステートからの到達可能なマクロステートが存在する場合について考える．その決定性ステートグラフの通りに各マクロステートに含まれる対応するステートを遷移してゆく時，実行の各ステップについてどんな要求に対しても対応する遷移が存在し，かつその遷移先で充足可能性を保存している．

4 MPI を用いた並列化手法

以上の手続きを，現実規模のリアクティブシステム動作仕様の充足可能性，及び段階的充足可能性の判定に用いるとき，その時間的なコストは莫大なものとなる．したがって，一台の計算機で現実規模の動作仕様の性質判定を現実的な時間内に行うことは困難である．本節では，前節までの各種手続きに対して，複数台の計算機を用いる実装手法を提案する．本稿では，代表的な並列計算用のライブラリである **Message Passing Interface** [10][12] (以下では MPI と記す) を利用した実装手法について述べる．

4.1 MPI の概要

メッセージパッシングはプロセス間のデータ移動をプログラム中に明示的に記述する実装手法であり，MPI はメッセージパッシングの標準 API となっている．MPI の特徴は，プログラマ自身が並列処理を明示的に記述できることである．また，MPI プログラムは同一のプログラムを持つ複数のプロセスが，それぞれ異なるデータに対して同時に実行され，その過程でメッセージ通信を行い得ることも特徴の一つである．これを **Single Program Multi Data**(以下では SPMD と記す) と呼ぶ．各プロセスには**ランク**と呼ばれる一意のラベルが設定され，単一のプログラム中に各プロセスの行うべき処理をそのランクにより明示的に記述することが可能である．SPMD のスタイルを取ることにより，プログラマはプロセスごとに固有の実装を用意する必要がない．

4.2 グラフ構成手続きの並列化実装

これまでに与えたリアクティブシステム動作仕様の充足可能性判定手続きについて，判定手続き全体の計算コストに対するグラフの構成手続きにかかる計算コストの割合を，すでに Java 言語により実装されている単一計算機による検証系 **Titech Temporal Tableau Engine**(以下では T^3E と記す) [7][6][8] を用いて計測した．その結果，付録 A であげた簡易 n 階建てエレベータシステムの例では，2 階建ての場合 96%，3 階建てでは 98%，さらに 4 階建ての動作仕様では 99%もの高い割合を占めていることが明らかとなった．

本節では，このグラフの構成手続きを MPI を利用することで並列化し，判定手続き全体の時間コストを抑える実装手法について述べる．手続き 3.2 においてサクセッサステートの集合を計算する箇所 (7–10 行目) についてみると，ステート s のサクセッサ集合を計算するために，他のステートの情報を全く必要としないことが分かる．そこで本稿ではこの部分を MPI を用いて並列に処理するステートグラフ構成手続きの並列化実装を以下で与える．

手続き 4.1 (ステートグラフ構成手続きの並列化実装) 本プログラムは，これから行う処理内容に応じてグラフを管理する graphMngr プロセスと，サクセッサ集合を計算する decomposer プロセスに分かれる．本プログラムを実際に複数の計算機を用いて実行する際には，利用する計算機のリストを用いてそれぞれの計算機に実行するプロセスを 1 つずつ割り当て，graphMngr プロセスを実行する計算機 1 台と decomposer プロセスを実行する複数台の計算機に分けて実行する．以下の擬似コードでは，MPI を利用する送受信には **by MPI** と付記した．

このプログラムは式 φ を入力とし，ステートグラフ $\langle V_S, v_0, E\rangle$ を出力する．

```
1:  parallelConstSGraph(φ : formula) {
2:      if (rank is graphMngr's rank)
3:          ⟨V_S, v_0, E⟩ := graphMngrJob(φ) ;
4:      else
5:          decomposerJob;
6:  }
```

```
1:  graphMngrJob(φ : formula) {
2:      v_0 := {φ};
3:      V_S := {v_0};
4:      E := ∅;
5:      V'_S := ∅;
6:      D_P := {d_P | d_P is a decomposer process} ;
7:      while (V_S − V'_S ≠ ∅) {
8:          foreach (s ∈ V_S − V'_S) {
9:              if (D_P ≠ ∅) {
10:                 d_P := get an element in D_P;
11:                 D_P := D_P − {d_P};
12:                 put (d_P, s) on proc_table ;
13:                 send s to d_P by MPI;
14:             }
15:             else
16:                 break;
17:         }
18:         receive (S : set of state) from d_P by MPI;
19:         V_S := V_S ∪ S ;
20:         s := get the value of d_P from proc_table ;
21:         foreach (s' ∈ S ) {
22:             E := E ∪ {⟨s, s'⟩} ;
23:         }
24:         remove (d_P, s) from proc_table ;
25:         D_P := D_P ∪ {d_P} ;
26:         V'_S := V'_S ∪ {s} ;
27:     }
28:     send termination_message
            to all decomposer processes by MPI;
29: }
```

```
1:  decomposerJob {
2:      while (true) {
3:          receive (s : state) from graphMngr by MPI;
4:          if (s = termination_message)  break;
5:          S := decomp(Temp(s));
6:          send S to graphMngr by MPI;
7:      }
8:  }
```

parallelConstSGraph
MPI を用いた実装は SPMD のスタイルを取るため，すべてのプロセスはまずこのメソッドを実行する．2–5 行目: プロセス固有のランクにより，プロセス毎に行う処理を分ける．

graphMngrJob
graphMngr プロセスの行う処理．まだ分解されてい

ないステートを decomposer プロセスに振り分け，その結果を収集しグラフを管理する．
6 行目: D_P は現在，分解作業を行っていない decomposer プロセスの集合を表す．
9–14 行目: 未処理のステートが残っている場合には，作業を行っていない decomposer プロセスが無くなるまでステートを分配する．
12 行目: 分解処理を割り当てた decomposer プロセスと，分解処理対象のステートをマップに保存．このマップのキーは decomposer プロセス，値はステートである．
24 行目: proc_table から d_P で処理したステートを取り出す．
28 行目: すべての decomposer プロセスに処理終了を知らせるメッセージを送信する．

decomposerJob
decomposer プロセスの行う処理．ステートを受信し，そのステートを分解した結果のステート集合を送信する．
4 行目: graphMngr から処理終了を知らせるメッセージを受け取るまで処理を続ける．

4.3 充足可能性判定

MPI を利用した充足可能性判定手続きの並列化実装は，上の手続き 4.1 を用いて以下のように実装される．

手続き 4.2 (充足可能性判定の並列化実装)
リアクティブシステムの動作仕様 $Spec = \langle \mathcal{R}, \mathcal{S}, \varphi \rangle$ を入力とし，*Spec* が充足可能であれば真を，そうでなければ偽を出力する．

1. G := *parallelConstSGraph*(φ) によりステートグラフ G を構成する．
2. graphMngr プロセスを実行した計算機により，G を極大強連結成分に分割する．ここで，自己充足となる部分グラフが存在するならば真を，そうでなければ偽を出力する．

4.4 タブローグラフ決定化の並列化実装

3.5 節で述べた通り，動作仕様の段階的充足可能性を判定するにはステートグラフの決定化が必要である．本節では段階的充足可能性判定の手続きを並列化実装するために必要な，ステートグラフ決定化手続きの並列化実装を与える．この並列化実装はステートグラフの決定化手続き 3.4 を，手続き 4.1 とほぼ同様の手法によって並列化したプログラムである．

手続き 4.3 (ステートグラフ決定化手続き並列化実装)
本プログラムは，これから行う処理内容に応じてグラフを管理する graphMngr プロセスと，サクセッサ集合を計算する sucComp プロセスに分かれる．

本プログラムはステートグラフ $G = \langle V_S, v_0, E \rangle$ と命題変数の集合 $\mathcal{P}$ を入力とし，$\mathcal{P}$ の要素からなる列に関して，G を決定化した決定性ステートグラフ $\mathcal{G} = \langle \mathcal{M}, \mathbf{m}_0, \mathcal{E} \rangle$ を出力する．

```
1:  parallelConvDet(G : state graph,
                    P : set of atomic propositions) {
2:    if (rank is graphMngr's rank)
3:      ⟨M, m0, E⟩ := graphMngrJob(G, P) ;
4:    else
5:      sucCompJob(P);
6:  }
```

```
1:  graphMngrJob(G : state graph,
                 P : set of atomic propositions) {
2:    m0 := {v0};
3:    M := {m0};
4:    E := ∅;
5:    M' := ∅;
6:    SC_P := {sc_P | sc_P is a sucComp process};
7:    while (M − M' ≠ ∅) {
8:      foreach (m ∈ M − M') {
9:        if (SC_P ≠ ∅) {
10:         sc_P := get an element in SC_P;
11:         SC_P := SC_P − {sc_P};
12:         put (sc_P, m) on proc_table ;
13:         send (m, suc(m)) to sc_P by MPI;
14:       }
15:       else
16:         break;
17:     }
18:     receive suc_table from sc_P by MPI;
19:     m := get the value of sc_P from proc_table ;
20:     foreach (p ∈ 2^P) {
21:       m' := get the value of p from suc_table ;
22:       if (m' ≠ ∅) {
23:         M := M ∪ {m'};
24:         E := E ∪ {⟨m, p, m'⟩};
```

```
25:          }
26:       }
27:       remove (sc_P, m) from proc_table ;
28:       SC_P := SC_P ∪ {sc_P};
29:       M' := M' ∪ {m};
30:    }
31:    send termination_message
             to all decomposer processes by MPI;
32: }

1:  sucCompJob (P : set of atomic propositions) {
2:     while (true) {
3:        receive (m : macrostate, S : set of state)
                from graphMngr by MPI;
4:        if (m = termination_message)  break;
5:        foreach (p ∈ 2^P) {
6:           if ((S \ p) ≠ ∅)
7:              put (p, (S \ p)) on suc_table;
8:           else
9:              put (p, ∅) on suc_table;
10:       }
11:       send suc_table
                to graphMngr by MPI;
12:       clear suc_table;
13:    }
14: }
```

parallelConvDet
すべてのプロセスはまずこのメソッドを実行する．
2–5 行目: プロセス固有のランクにより，プロセス毎に行う処理を分ける．

graphMngrJob
graphMngr プロセスの行う処理．まだ処理されていないマクロステートを sucComp プロセスに振り分け，その結果を収集しグラフを管理する．
6 行目: SC_P は現在処理を行っていないプロセスの集合を表す．
18 行目: sucComp プロセスから命題変数の集合と，マクロステートのマップを受信する．このマップのキーは命題変数の集合，値はマクロステートである．

sucCompJob
sucComp プロセスの行う処理．マクロステートとそのマクロステートに含まれるステートの遷移先を集めた集合を受信し，受信したマクロステートの遷移先となるマクロステートの集合を計算する．またその結果をマップとして返信する．
5–9 行目: 命題変数集合をキー，マクロステートを値とするマップの作成を行う．
12 行目： 返信用のマップを初期化する．

4.5 段階的充足可能性判定

MPI を利用した段階的充足可能性判定手続きの並列化実装は，上記の手続き 4.1， 4.3 を用いて以下のように実装される．

手続き 4.4 (段階的充足可能性判定の並列化実装)
リアクティブシステムの動作仕様 $Spec = \langle \mathcal{R}, \mathcal{S}, \varphi \rangle$ を入力とし，*Spec* が段階的充足可能であれば真を，そうでなければ偽を出力する．

1. G := *parallelConstSGraph*(φ) により，ステートグラフ G を構成する．
2. ステップ 1. において graphMngr プロセスを実行した計算機により，以下のステップ (a)–(c) を G が変化しなくなるまで繰り返す．
   (a) イベンチャリティ式が含まれ，かつそのイベンチャリティ式が満たされるステートに到達不可能なステートを削除する．
   (b) 削除されたステートに入るエッジ，及び削除されたステートから出るエッジをすべて削除する．
   (c) G の開始ステートが削除された場合には，偽を出力する．
3. ステップ 2. により変更された G を手続き 4.3 により決定化し，決定性ステートグラフ $\mathcal{G}$:= *parallelConvDet*($G, \mathcal{R} \cup \mathcal{S}$) を構成する．
4. 以下の操作を，ステップ 3. において graphMngr プロセスを実行した計算機を用いて，$\mathcal{G}$ が変化しなくなるまで繰り返す．
   (a) $\mathcal{G}$ 中の行き止まりのマクロステートを削除する．
   (b) 削除されたマクロステートに入るエッジ，及びそのマクロステートから出るエッジをすべて削除する．
5. $\mathcal{G}$ の開始マクロステートが削除された場合に偽を，そうでなければ真を出力する．

		T^{3}EM(decomposer プロセス数)										
n	T^{3}E	2	3	5	7	9	11	13	15	17	19	21
2	395	1835	1755	1490	1456	2374	1859	1979	3711	2225	1778	1841
3	2368	5344	4602	4470	3954	3745	3981	3669	3882	3629	4107	4288
4	47824	45983	34482	24620	21725	19237	19083	19080	18841	18836	18139	18411

表 1: n 階建てエレベータの例による比較 (単位:ms)

	decomposer プロセス数 (d)										
	2	3	5	7	9	11	13	15	17	19	21
T^{3}EM	45983	34482	24620	21725	19237	19083	19080	18841	18836	18139	18411
T^{3}E/d	23912	15941	9565	6832	5314	4348	3679	3188	2813	2517	2277
差	23912	15941	15055	14893	13923	14735	15401	15653	16023	15622	16134

表 2: T^{3}E の計測時間を decomposer プロセスの数 d で割った値との比較 (単位:ms)

5 実験結果と評価

本節ではすでに Java 言語により実装されている単一計算機による検証系 T^{3}E[7][6][8] を拡張して MPI を利用する充足可能性判定手続き 4.2 を実際に実装し，拡張前の T^{3}E と比較することでその有効性を確認する．本稿で提案したプログラムを Java 言語によって実装するにあたり，Java に対する MPI 実装である **mpiJava** Version 1.2.5[10] を利用した．以下では，拡張前の検証系 T^{3}E と区別するため，拡張後の検証系を **Titech Temporal Tableau Engine for Multiprocessor** (以下では T^{3}EM と記す) と呼ぶ.

5.1 簡易 n 階建てエレベータシステムの動作仕様による比較

ここでは T^{3}E, T^{3}EM 共に，入力として付録 A にあげた簡易 n 階建てエレベータシステムの動作仕様を与え，充足可能性判定の終了までにかかる実行時間を計測し比較する．測定に利用したすべての計算機の実行環境は同一で以下の表 3 の通りである．この測定結果を表 1–2 に示す．T^{3}EM の実行時のプロセスの数は graphMngr プロセスと decomposer プロセスの数の総和となり，1 つのプロセスに対して 1 台の計算機を割り当てる．また graphMngr プロセスは各実行時に 1 つしか存在しない．すなわち，decomposer プロセスの数が 3 となっている計測結果は，graphMngr プロセスを実行する計算機を 1 台と decomposer プロセスを実行する計算機を 3 台用いた実行による計測結果を表す．表 2 は，4 階建てのエレベータシステムの例について，T^{3}EM の実行時間と，T^{3}E の実行時間を単純に比較対象の decomposer プロセスの数で割ったものを，比較した表である．

CPU:	AMD Opteron 2GHz
RAM:	2GBytes
OS:	RedHatLinux 9.0
Java:	Version 5.0
mpiJava:	Version 1.2.5

表 3: 実験に利用した計算機環境

5.2 時間効率に対する評価

表 1 の結果から，入力されるリアクティブシステムの動作仕様が複雑であるほど提案する実装の効果がはっきりと現れてくることが確かめられる．これは，複雑な動作仕様に対する性質判定の時間的なコストを引き下げるという目的に対して，本実装が確かにその目的を実現できることを示している．

また表 2，図 4 から 4 階建てエレベータの充足可能性判定の例について，T^{3}E の実行時間を単純に比較対象の decomposer のプロセス数で割った値と T^{3}EM の実行時間との差が，decomposer プロセスの数が 5 となるあたりから 11 程度までにおいて，収束していることが分かる．この結果は，ある程度の台数を用意すれば，本稿で提案する並列化手続きが

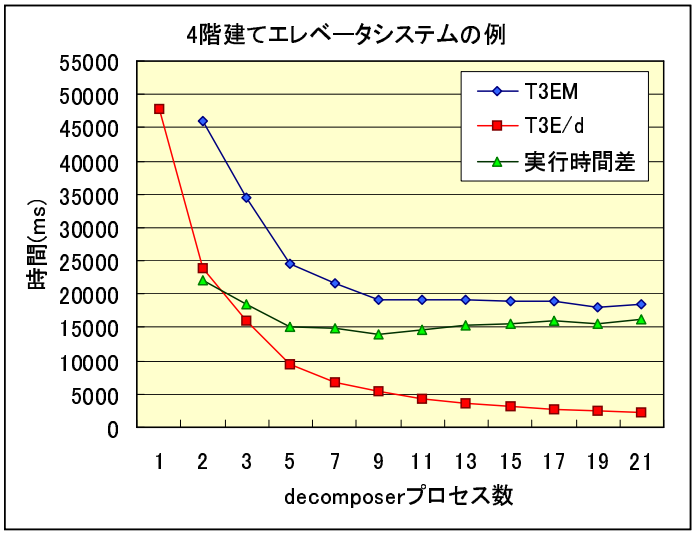

図 4: 表 2 を表すグラフ

n	ステート数 (個)
2	206
3	6353
4	88454
5	1280924

表 4: n 階建てエレベータの例から構成されるタブローグラフのサイズ

台数分の効果を示すことをあらわしている．台数分効果があらわれる一方で，decomposer プロセス数が少ない場合に，単純に decomposer プロセス数で割った値と差が開いている事に関しては，例えば，次のような処理割り当てのオーバーヘッドの存在が大きいと考えられる．graphMngr プロセス側には，decomposer プロセスに割り当てるべきまだ未処理のステートが存在するにもかかわらず，decomposer プロセスに空きがない場合，graphMngr プロセスは待ち状態に入ってしまう．このオーバーヘッドの様子を図 5 に示す．また 2 階建て，及び 3 階建てエレベータ仕様についても表 1 に示した計測結果を用いて表 2 と同様の比較を行う．このとき，ある数以上の decomposer プロセスが存在すれば，処理割り当てのオーバーヘッドが減少していき，実験結果をある収束した値に抑えることが可能である．この状態を図 6 に示す．

これは，図 5， 6 を比較した時に，実行時間の差として,「相手待ち」と,「通信時間」が大きく関係していることからも分かる．この部分のオーバーヘッドが decomposer プロセスが少ない場合に顕著に現れるため，図 4 における decomposer プロセス数が少ない時に実行時間の差が大きく離れている現象が現れると考えられる．また，図 4 の decomposer プロセスが 11 台以上の結果は，実行時間差は大きくなっていくが，T^{3}EM の実行時間がある一定の値に収束してくる．これは，図 6 の状態となるのに必要な台数が揃い，graphMngr プロセスにおける処理時間が最小まで減少した状況になっていることが理由だと考えられ，これ以上台数を用意しても，本並列化手法では効果が得られないことが分かる．

そこで，本実装において decomposer プロセスを何台用意すれば十分か，という問題が浮かんでくる．十分な台数とはすなわち，図 5 にあらわれる様なオーバヘッドが減少し，図 6 の状態になるのに必要な decomposer プロセス数のことを言う．これは以下の式から計算が可能である．

$$(\text{最低限必要な decomposer プロセス数}) = \left\lceil \frac{(1\ \text{ステートの展開にかかる時間})}{(1\ \text{ステートの送信準備に必要な時間})} \right\rceil$$

1 ステートの展開にかかる時間は graphMngr プロセスにとっては，待ち時間となる．1 つのステートの展開を待つ待ち時間を最大限に活用した状態では，待ち時間の間に他のステートを次々と別の decomposer プロセスへ割り当てていく事が可能である．上式の

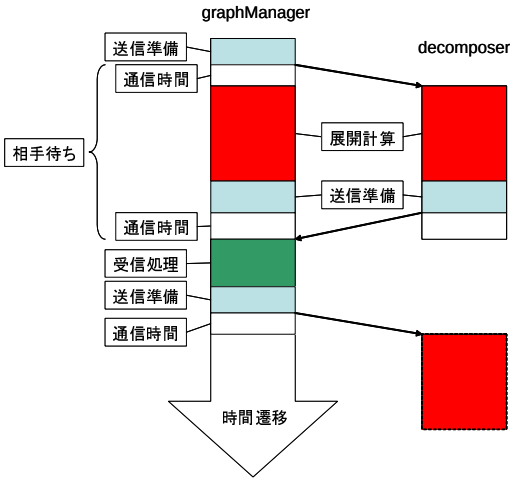

図 5: decomposer 1 台と graphMngr 1 台の時におけるオーバーヘッドを表した図

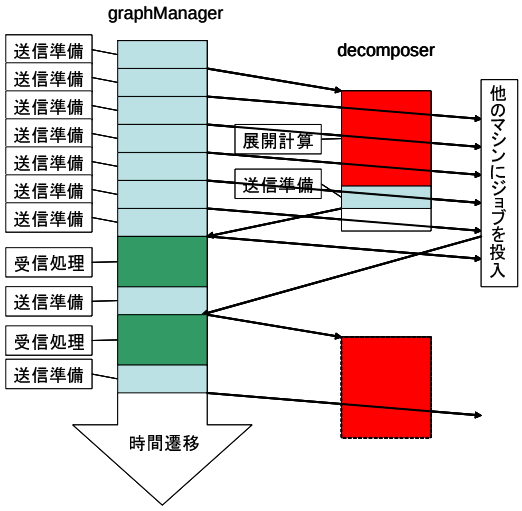

図 6: decomposer 1 台と graphMngr *n* 台の時における最低限のオーバーヘッドを表した図

正当性は,「相手待ち」の時間を全てジョブ投入の時間で埋める事が出来れば，実際の実行時間は送受信の準備にかかる時間のみとなるため，グラフの構成にかかる時間を最小にする事が出来ることから示せる．また，実際には，graphMngr プロセスが 1 台必要なので，全体で利用するマシン数は (上式) + 1 となる．例えば，4 階建てエレベータ仕様の実験結果から，概算によりこの計算を行うと，最低限必要な decomposer プロセス数は 8 という結果となり，図 4 の実験結果が収束しはじめる地点とほぼ同じ値となる．実験を行える場合は，上式の計算に必要な情報はすべて実験値から得られる．また，ステートの展開時間や送信準備時間にはステートの持つ情報量が大きく関係している．そこで，実験を行えないような仕様に対しては，現在まで得られている結果を利用し，仕様の大きさからそれらの値を計算する事で，ある程度の概算を得る事が可能であると考えられる．

5.3 メモリ効率に対する評価

図 7， 8，9 の結果は，それぞれ Java5.0 の機能を用いモニタリングした，実行時における T^{3}E と T^{3}EM の graphMngr プロセス，decomposer プロセスのメモリ使用率の遷移を表している．この結果から，T^{3}E に対して，T^{3}EM の graphMngr プロセス，decomposer プロセスのいずれも，メモリ使用率が減少している事が分かる．これは本来，T^{3}E では単一マシンで行っていた作業を，それぞれに分担させ

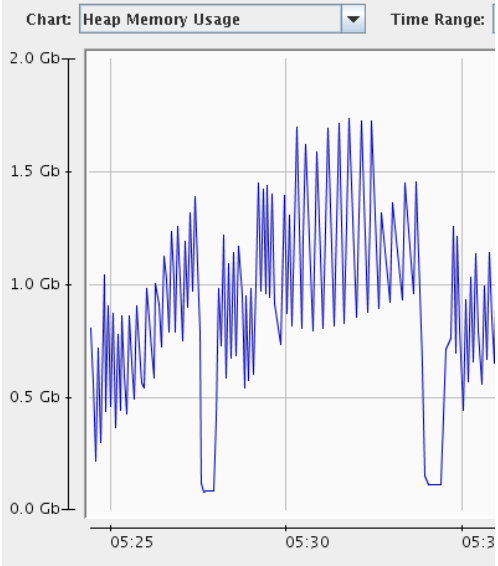

図 7: T^{3}E のメモリ使用率の遷移を表すグラフ

た結果であり，メモリ効率の点においても，本実装が効果を発揮していることを示している．しかし，現在の本実装では，検証を行える仕様の大きさが，T^{3}E と差がほとんどないことが，他の実験から確かめられている．これは，図 7 側に現れている急激なメモリ開放が，図 8 や 9 には現れていない事が原因だと考えられる．これは本実装では，プロセス間の通信のために用意するデータ中の，グラフを保持する際に不必要なデータをガーベッジコレクションしていないことが原因である．つまり，本実装では，

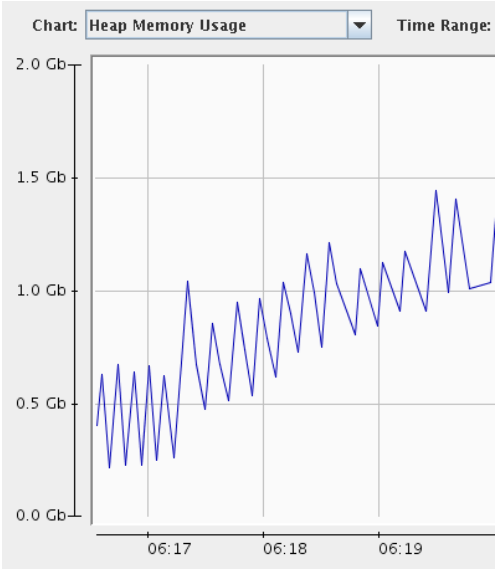

図 8: T[3]EM の graphMngr プロセスのメモリ使用率の遷移を表すグラフ

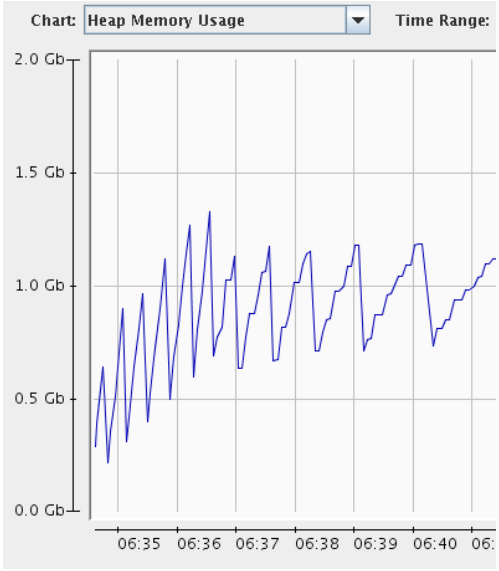

図 9: T[3]EM のある decomposer プロセスのメモリ使用率の遷移を表すグラフ

まだ十分にメモリ効率が上がっていないが，上がる見込みがあることが分かる．

図 7 の急激にメモリ使用率が減少している点に注目すると，タブロー法における各種性質判定の際に必要なグラフの情報は，この部分の急激に減少しきった点程度のメモリしか必要としていない事が分かる．すなわち，次ステートの計算などで利用するグラフを構成していく際に必要なメモリが，プログラム全体で必要とするメモリの大半を占めている事がわかる．この事から，本実装では上述の次ステートの計算を分散させて行う事から，ガーベッジコレクションを厳密に行えば，従来のエンジンである T[3]E に比べ，より大きな仕様に対し検証が可能と考える．以上から，メモリ効率に関しても本並列化手法は非常に効果的である．

6 まとめと今後の課題

本稿では，時間論理で記述されたリアクティブシステムの動作仕様の性質のうち，充足可能性と段階的充足可能性の判定手続きに対する並列分散処理を用いた実装手法について述べた．ステートグラフ構成手続きにおいてステートのサクセッサ集合の計算が，他のステートの情報を必要とすることなく行えることに着目し，この箇所を並列に処理する MPI を利用したグラフ構成手続きの実装手法を与えた．段階的充足可能性の判定に必要となるステートグラフの決定化手続きについても，ステートグラフ構成手続きとほぼ同様の点に着目し，MPI を利用した決定化手続きの実装手法を与えた．そして，これらの MPI を利用した実装を用いた，充足可能性および段階的充足可能性の並列分散処理による判定手続きを示した．

また，本稿では提案した充足可能性判定の並列化実装手法について，実際に Java 言語により実装し，本実装が大きな動作仕様に対して有効であることを実験により確かめた．さらに，より複雑な動作仕様に対しては本稿で提案した手法により，その充足可能性判定の時間コストを，1 台の計算機を用いた場合の時間コストをサクセッサ集合の計算を行う計算機の台数で割った値にまで抑えられることが，実験測定値から予測される．また，メモリコストに対しても実験測定値から，本稿で提案した手法により，1 台の計算機を用いた場合のコスト以下に抑えられていることを確認し，今後の展望として従来よりも大きな仕様に対して検証を行うことが可能だと予測される．

最後に今後の課題として，以下の様なものが挙げられる．

- 本稿で与えた並列化手法の実装における，ガーベッジコレクションの強化
- オーバーヘッドを抑えるため，送受信するデー

タの大きさやその内容，プロセスの待ち状態を最小限に抑える処理割り当てなどについての考察.

- 本稿で与えた並列化した段階的充足可能性判定手続きの Java 言語による実装と，強充足可能性判定の並列化実装手法の完成.
- 高速化された分の時間を若干犠牲にし，ディスクメモリ利用による大規模な仕様に対する検証エンジンの実装
- 例えば構成するタブローグラフも分散して保持するといった，分散化におけるメモリ空間に対する効果的な実装
- 動作仕様の部分評価 [6] を並列に行うことによる高速化手法の導入.

参考文献

[1] 萩原 茂樹, 米崎 直樹:Muller オートマトンを用いたリアクティブシステムの仕様検証法とその完全性, 第二回システム検証の科学技術シンポジウム 予稿集 (2005),pp.52–63.

[2] Noriaki Yoshiura:Decision Procedures for Several Properties of Reactive System Specification,Software Security – Theories and Systems,Lecture Notes in Computer Science,No.3233(2004),Springer,pp.154–173.

[3] 島川 昌也:時間論理仕様によるリアクティブシステムの実行時検証と仕様の性質に関する研究, 平成 17 年度 修士論文 (2006), 東京工業大学大学院情報理工学研究科 計算工学専攻.

[4] 森 亮靖, 友石 正彦, 米崎 直樹:時相論理によるリアクティブシステム仕様の実現可能性に関する分類, コンピュータソフトウェア, Vol.15,No.3(1998),pp.25–37.

[5] 森 亮靖:リアクティブシステムの実現可能性に関する研究, 博士論文 (1994), 東京工業大学大学院理工学研究科 計算工学専攻.

[6] 青島 武伸, 米崎 直樹:リアクティブシステム仕様の検証系に関する研究, 平成 14 年度 博士論文 (2003), 東京工業大学大学院情報理工学研究科 計算工学専攻.

[7] 青島 武伸, 米崎 直樹:時間論理によるリアクティブシステム仕様の検証の効率化, コンピュータソフトウェア,Vol.20,No.3(2003),pp.30–53

[8] 速川 徹:効果的な結果表示を行う検証器インターフェイスの実装に関する研究, 平成 14 年度 修士論文 (2003), 東京工業大学大学院情報理工学研究科 計算工学専攻.

[9] 佐藤 正崇:実現不能なリアクティブシステム仕様の原因解析に関する研究, 平成 15 年度 修士論文 (2004), 東京工業大学大学院理工学研究科 国際開発工学専攻.

[10] Message Passing Interface http://www-unix.mcs.anl.gov/mpi/

[11] The HP Java Project http://www.hpjava.org/index.html

[12] P. パチェコ 著, 秋葉 博 訳:MPI 並列プログラミング (2001), 培風館.

[13] 安藤 崇央, 大滝 大輔, 米崎 直樹:時間論理タブロー証明器の MPI による実装, 第二回システム検証の科学技術シンポジウム 予稿集 (2005),pp.227–245

[14] 大滝 大輔:時相論理による仕様の並列検証に関する研究, 平成 16 年度 学士論文 (2005), 東京工業大学大学院情報理工学研究科 計算工学専攻.

[15] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen and P. McKenzie:Model-Checking Techniques and Tools, System and Software Verification(2001),Springer-Verlag.

[16] Howard Wong-Toi and David L. Dill:Synthesizing Processes and Schedulers from Temporal Specifications, CAV '90: Proceedings of the 2nd International Workshop on Computer Aided Verification(1991),Springer-Verlag,pp.272–281.

[17] Amir Pnueli and Roni Rosner:On the Synthesis of a Reactive Module, POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages(1989),ACM Press,pp.179–190.

[18] Amir Pnueli and Roni Rosner:On the Synthesis of an Asynchronous Reactive Module, ICALP '89: Proceedings of the 16th International Colloquium on Automata, Languages and Programming(1989),Springer-Verlag,pp.652–671.

A 付録

簡易 n 階建てエレベータシステムの動作仕様 $\langle \mathcal{R}, \mathcal{S}, \varphi \rangle$ を記述した例を以下に示す．ここで $\mathcal{R}$ は要求変数，$\mathcal{S}$ は応答変数のそれぞれ集合である．φ は以下に示す式集合 φ' のすべての要素の連言である．

$\mathcal{R}$ = {

$LocBtn(i)(i = 1..n)$,	//i 階の呼出しボタンが押された
$OpenBtn$,	//「開く」ボタンが押された
$CloseBtn$	//「閉じる」ボタンが押された

}

$\mathcal{S}$ = {

$Loc(i)(i = 1..n)$,	//リフトが i 階にある
$ReqL(i)(i = 1..n)$,	//リフトが i 階に行く要求がある
$Open$,	//ドアが開いている
$Movable$,	//リフトが可動状態である
$OpenTimedOut$,	//「開く」ボタンの時間切れ
$ReqOpen$	//ドアを開く要求がある

}

φ' = {

//リフトはどこかの階にある

$\Box(\bigvee_{1 \le i \le n} Loc(i))$,

//リフトがある階にある場合，他の階にはない

$\Box(\bigwedge_{1\le i\le n}(Loc(i) \rightarrow \bigwedge_{j=1..n, i\neq j} \neg Loc(j)))$,

//呼び出しボタンが押されればその階にいつか行く
//かつその要求が満たされるまで要求し続ける
$\Box(\bigwedge_{1\le i\le n}(LocBtn(i) \rightarrow \Diamond Loc(i) \wedge [Loc(i) \wedge ReqL(i)]ReqL(i)))$,

//リフトのある階に要求があればドアを開く
//ドアが開いている間，リフトは停止
$\Box(\bigwedge_{1\le i\le n}(Loc(i) \wedge ReqL(i) \rightarrow Open \wedge [Movable]Loc(i)))$,

//呼び出しボタンが押されるまで要求は出さない
$\Box(\bigwedge_{1\le i\le n}(Loc(i) \wedge Movable \rightarrow [LocBtn(i)]\neg ReqL(i)))$,

//その階に要求がなければ，ドアは開かない
$\Box(\bigwedge_{1\le i\le n}(Loc(i) \wedge \neg ReqL(i) \rightarrow \neg Open))$,

//途中階を通る
($n \geq 3$ の場合)
$\Box(\bigwedge_{\substack{1\le i\le n-2\\ 3\le j\le n\\ i\le j-2}}(Loc(i) \wedge ReqL(j) \rightarrow \bigwedge_{i+2\le k\le j}\langle Loc(k)\rangle Loc(k-1)))$,

($n \geq 3$ の場合)
$\Box(\bigwedge_{\substack{1\le i\le n-2\\ 3\le j\le n\\ i\le j-2}}(Loc(j) \wedge ReqL(i) \rightarrow \bigwedge_{i+2\le k\le j}\langle Loc(k)\rangle Loc(k+1)))$,

//ドアの開閉とリフトの可動状態の関係
$\Box(Open \rightarrow [\neg Open]\neg Movable)$,
$\Box(\neg Open \rightarrow [Open]Movable)$,

//ドアの開放状態には時間制限がある
$\Box(Open \rightarrow \Diamond OpenTimedOut)$,

//時間制限内で “開く” ボタンを押せば
//ドアの開放を要求
$\Box(OpenBtn \wedge \neg OpenTimedOut \rightarrow ReqOpen)$,

//ドア開放時間制限が過ぎたらドアを閉める
$\Box(OpenTimedOut \rightarrow \neg Open)$,

//開放要求がなくかつ “閉じる” ボタンが押されれば
//ドアを閉じる
$\Box(CloseBtn \wedge \neg ReqOpen \rightarrow \neg Open)$,

//ドア開放要求が真，可動状態が偽ならドアを開く
$\Box(ReqOpen \wedge Movable \rightarrow Open)$
}