論文 / 著書情報
Article / Book Information

題目(和文)	リアクティブシステム仕様の検証方法とその実装に関する研究
Title(English)	
著者(和文)	安藤崇央
Author(English)	Takahiro ANDO.
出典(和文)	学位:博士(工学), 学位授与機関:東京工業大学, 報告番号:甲第9587号, 授与年月日:2014年4月30日, 学位の種別:課程博士, 審査員:米崎 直樹,徳田 雄洋,権藤 克彦,渡部 卓雄,西崎 真也
Citation(English)	Degree:Doctor (Engineering), Conferring organization: Tokyo Institute of Technology, Report number:甲第9587号, Conferred date:2014/4/30, Degree Type:Course doctor, Examiner:,,,,
学位種別(和文)	博士論文
Type(English)	Doctoral Thesis

平成 25 年度 博士論文

リアクティブシステム仕様の検証方法とその実装に関する研究

九州大学 大学院システム情報科学研究院
情報知能工学部門
安藤 崇央

平成 26 年 2 月

指導教員
東京工業大学
大学院情報理工学研究科 計算工学専攻
米崎 直樹 教授

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　研究の背景と目的

高い安全性が求められるシステムについては、その開発工程のどの段階においても誤りが混入しないよう、常に確認を行いながら開発を進める必要がある。例えば、仕様書を基に設計したシステムのモデルが正しく要求仕様を満足しているか、実装された個々のモジュールやシステム全体が期待通りに振る舞うかなど、各工程の成果物についてその都度誤りが混入していないか確認する必要がある。

しかし、宇宙開発機器やロボットなど、要求仕様や動作仕様が高度に複雑なリアクティブシステムに対してこのような確認作業を人手で行うことは、確認作業が膨大になりその作業コストが大きくなるだけではなく、確認ミスや確認漏れなどにより、その確認の信頼性が失われる場合もある。そこで、モデル検査やモデルベーステストなど、形式的手法を用いてこれらの確認作業をサポートする技術についての研究がなされ、開発現場に取り入れられ始めている。

モデル検査 [25] は、対象となるシステムを状態遷移システムなどとしてモデル化し、そのモデルが満たすべき要求を満足するかを判定する技術である。モデル検査器は、網羅的に遷移をたどることによってモデルが要求を満足するか判定するツールで、要求を満足しない場合には、その証拠となる振る舞いを反例として出力する。代表的なモデル検査器としては、Holzmann の開発した SPIN [23] や Cimatti らによる NuSMV [12] などがよく知られている。

モデルベーステストは、システムのモデルからテスト工程に用いるテストケースを導出する技術で、モデルから実行トレースを探索する、Liveness 性を確認するテストの生成にモデル検査の反例パスを利用する、モデルを同値分解した上でテストケースを選択するなど、形式的手法を用いてテストケースを導出する手法も提案されている。

一方でシステムが満たすべき要件そのものの品質を上げる技術が研究

されている。これは仕様獲得という上流工程において、できるかぎり正確で漏れのない記述を得ておくことは後の開発プロセスの効率を大きく向上させるという事実に基づいている。

その一つとして、意味論が数学的に厳密に定義された形式的な言語を用いて要件を記述する形式仕様記述がある。形式的な仕様記述言語を用いることで、システム要件から曖昧性を排除し、技術者間の要件解釈の齟齬を防ぐことができる。

また、これらの言語を仕様記述に用いる利点として、仕様そのものの形式的な検証が可能となる点が上げられる。仕様に対する検証は主に開発工程の初期段階で実施されるため、形式的仕様検証による仕様の不備の早期発見は、開発工程の大幅な手戻りを防ぐ効果がある。

形式的仕様記述に用いられる言語としては、計算木論理 (Computation Tree Logic, CTL) [19] や線形時相論理 (Linear Temporal Logic, LTL) [33] などの時相論理式やZ[35], B [2], VDM-SL [24] などのモデリング言語、代数的な仕様記述言語としてLOTOS [20] やCafeOBJ [32] などが知られている。

本論文では、動作仕様の制約的な記述が可能でモデル検査等で一般的に用いられる線形時相論理を、仕様記述言語として用いる。また、本論文では線形時相論理を用いて記述されるリアクティブシステム仕様の仕様検証技術の実用化に向け、効率の良い検証手法の研究やその実装に関する研究を行う。

システムの要求定義者は、そのシステムが満たすべき要件を動作仕様として記述する。もちろん要求定義者は、すべての要件を満たすシステムが存在し、構築できると信じて仕様を記述する。しかしシステムの規模が大きくなり動作仕様が複雑になると、単なる記述ミスや、要求定義者が想定しないケースで複数の要件が衝突するなどして、それらすべての要件を満たすシステムを実現できないこともあり得る。

リアクティブシステムのようなシステムを考えた場合、その動作仕様については矛盾した記述が存在しないことを意味する**充足可能性**（**無矛盾性**）はもちろんのこと、どのような入力列に対しても仕様を満足したままで適切に動作できるシステムが存在しうることを表す**実現可能性** [1][34] についても、満足されていなければならない。

Bloemらは、検証対象の仕様からシステムと環境の間の2人のプレーヤのゲームを構成し、そのゲームの勝敗判定にモデル検査器NuSMVを用いることで実現可能性判定器を実現したRAT [9][10] を開発している。

動作仕様の実現可能性判定の計算量は、一般に仕様の大きさに対して 2-EXPTIME-complete [39] であり、現実規模の仕様について実現可能性を判定することは困難である。そのため、仕様検証の実用化には判定にかかるコストを抑える手法が必要となる。

仕様検証のコストを抑える手法の一つとして、森らは実現可能性に比べて判定コストの小さい**段階的充足可能性**などの性質を、その判定方法とともに [55] で提案している。段階的充足可能性は “いかなる有限長の入力に対しても、充足性を保存するようにふるまうシステムが存在する” ことを表す。[55] にて提案されたこれらの性質は、いずれも実現可能性の必要条件となっている。動作仕様が実現可能であるかを判定する前に、実現可能性に比べ判定コストの小さいこれらの性質を満足するかどうか確認することは実際上有用である。

森らにより与えられた各性質の判定方法は、**タブロー法**に基づいている。タブロー法では**タブローグラフ**と呼ばれるグラフを構成する。リアクティブシステム仕様から構成されたタブローグラフは、仕様を満たすシステムの状態遷移モデルの集合を表している。このグラフを解析することで、動作仕様の各性質を判定することができる。

青島らは、その判定法を基に仕様の性質判定器を実装し、その効率化についても研究を行っている。青島らの実装では、判定手続中で構成するグラフの大きさを小さく抑える**プレステートグラフ表現** [52] や、グラフの構成手続きに二分決定グラフ *Reduced Ordered Binary Decision Diagram* [37][11] (以後、*BDD* と表記) を用いる効率化 [3] などが利用されている。これにより、自動検証可能な仕様の規模の拡大や、検証時間の短縮化がなされた。

しかし、それでもなお現実規模の仕様の性質判定を行うことを考えると、プレステートグラフ表現を用いたとしても構成すべきグラフが巨大になり、その構成にかかる時間的コストやグラフを保持するための空間を確保できないなどの理由で、1 台の計算機の能力では足りなくなる。

そこで、本論文ではこれらの問題に対応する手法として、処理の並列化やデータの分散化について研究を行った。しかしここで取り扱う問題が行列演算のように簡単に分割処理できるようなデータ構造をしておらず、さらに、その分割すべきデータが手続き中で動的に生成されるため、その実装には巧妙な工夫が数多く必要となる。また、それら工夫の組み合わせ方についても、単純な組み合わせで効率化が達成できるかどうかについては自明ではなく、研究する意義は大きいと考える。

モデル検査の分野においても処理の並列化や分散メモリを利用しようとする研究 [29][5] が存在し、モデル中のプロセスの振る舞いを表す状態遷移系同士の積や、そのシステム全体を表す状態遷移系と時相論理を用いて表現された検査式を表現する状態遷移系との積をあらわすグラフを分散メモリ上に構成し、到達可能性判定や検査式の検証を並列に行うモデル検査手法が提案されている。しかし、これらの検査器においても、取り扱う検査式は極めて小規模なものが前提とされており、その検査式からの状態遷移系の構成については分散化の対象外となっている。

タブロー法による仕様検証では、モデル検査で扱う検査式に比べ大規模な論理式を対象とし、その論理式からタブローグラフを構成する必要がある。本論文では、この “論理式からタブローグラフを構成する” という箇所について並列化や分散化の対象とする。

1.2 本論文の概要

本論文では、線形時相論理で記述されたリアクティブシステム仕様の充足可能性、および段階的充足可能性判を判定する性質判定器の実装手法について、研究したことを報告する。

まず、仕様の段階的充足可能性について、タブロー法を用いた完全な手続きと、その実装法について述べる。森らによって与えられていた従来の段階的充足可能性判定手続き [54] は、段階的充足可能な仕様を段階的充足不能と判定してしまう恐れのある手続きであった。そこで本論文では、萩原らによって与えられた決定性オートマトンを用いた判定法 [53] をタブロー法に応用した手続きを与える。本論文で与える段階的充足可能性判定手続きは、[51][54] で示されている判定手続きにタブローグラフの決定化手続きを追加し、決定化後のタブローグラフに対して行き止まり判定を行う手続きである。

さらに、その段階的充足可能性判定手続きの実現法の一つとして、終端ノードが特殊な二分決定グラフ $\mathrm{BDD_{PS}}$, $\mathrm{BDD_{MPS\_r}}$ を導入し、これらの二分決定グラフをタブローグラフの決定化手続きと行き止まり判定手続きに利用する実装手法を提案する。タブローグラフ中の各ノードについてそのノードからの遷移に関する情報を $\mathrm{BDD_{PS}}$ として表現し、それらの $\mathrm{BDD_{PS}}$ 同士の和集合演算を利用することによりタブローグラフの決定化手続きを実現する。また、決定性タブローグラフ中の各ノードに対してそのノードの遷移情報を $\mathrm{BDD_{MPS\_r}}$ として表現し、その $\mathrm{BDD_{MPS\_r}}$ の終

端ノードに空集合が含まれているか確認することで行き止まり判定手続きを実現する。$\mathrm{BDD_{PS}}$ や $\mathrm{BDD_{MPS\_r}}$ では共通部分を共有するため遷移情報を効率よく保持できる。また、その終端ノードを調べるだけで次ノード取得や行き止まり判定などが簡単に行えるという特徴も持つ。

次に、性質判定器の処理速度向上のため判定手続きの並列化について考察する。まず、性質判定手続きの要であるタブローグラフ構成手続きに注目し、この手続きの並列化手法について考察する。その結果、タブローグラフ中のあるノードからその遷移先のノード集合の作成を行う、次ノード作成処理がノード毎に処理できることを発見し、この処理を複数のノードについて同時に処理することで並列化できることを突き止めた。本論文では、*Message Passing Interface*(MPI) を用いて複数のノードに対する次ノード作成ステップを複数の計算機ノードにて並列に処理するタブローグラフ構成手続きの並列化実装法を与える。

本論文で与えるタブローグラフ構成手続きの並列化実装は、次のような特徴を持つ。

- 本実装手法による処理系は、構成するタブローグラフを一括管理するプロセスと、タブローグラフの次ノード計算を専門に行う複数のプロセスからなる。

- グラフ管理プロセスは、構成中のタブローグラフについての情報をすべて保持し、次ノード作成ステップが完了していないノードの情報を次ノード計算を専門に行うプロセスに送信する役割と、その結果を受信してグラフ情報を更新する役割を担う。

- 次ノード計算を専門に行うプロセスは、グラフ管理プロセスから受信したノード情報を基にそのノードの次ノード集合を計算し、結果をグラフ管理プロセスに返信する。

この実装は、タブローグラフ構成にかかる時間的なコストを抑えることができるが、グラフ情報が集中するため、グラフ管理プロセスを実行する計算機に空間的な負荷が集中してしまう。

そこでこれに対処するため、本論文ではタブローグラフ構成手続きの二つ目の実装手法として、タブローグラフを表すデータ構造と分散オブジェクト技術に注目し、グラフ情報を表すデータを分散配置することで実行環境上のメモリ空間の制限を抑え、構成できるタブローグラフの大きさの制限を緩和することを目的とする並列分散化手法を与える。その並列分散化実装は次のような特徴を持つ。

- MPI を用いた実装同様、個々のノードに対する次ノード計算を別々のプロセスで処理する。
- タブローグラフの情報は 1 箇所に集めず、次ノード計算を行うプロセスが自身の担当するノードを保持し、環境全体でタブローグラフの情報を分散保持する。
- タブローグラフのノード間の接続情報は、異なるプロセス間のオブジェクト同士の接続という形で実現する。

二つ目に提案する実装は以上のような特徴を持つため、逐次実行による処理系に比べ判定にかかる時間を短縮し、グラフ保持のための空間的な制約を軽減できる。

本論文では、本実装を分散オブジェクト技術を利用して実装する方法を与える。分散オブジェクト技術は、リモートに存在するオブジェクトをあたかもローカルに存在するかの様に扱うことができ、“グラフ中のノードとその次ノード” の関係のように互いに結びつきの強いオブジェクト同士を異なる計算機上に配置するのに適している。

さらに本論文では、タブローグラフ構成手続きの他に、段階的充足可能性判定に用いる二つの内部手続き、グラフ決定化手続き、および行き止まり判定手続きについての並列分散化手法についても与える。これら二つの内部手続きの並列分散化実装は、分散化されたタブローグラフに対して適用する実装となっている。

また、本論文では以上の並列化実装に加えて、タブローグラフ構成手続きの内部手続きである、式の分解手続きの実装法についての考察も行う。そこでは、式の分解手続きを命題論理の充足可能性問題[1]（SAT 問題）へと帰着し、この問題の解を SAT solver を用いて求めることで式の分解手続きを実現する手法について述べる。

SAT solver は命題論理の充足可能性問題を高速に解決することのできるツールで、その高速化に関する研究が多くなされている。有界モデル検査や論理回路のテスト用データの生成、スケジューリング問題などの多様な問題が、命題論理の充足可能性問題に帰着され、その解決に SAT solver が利用されている [8][28][16]。ここでは、タブローグラフ構成手続きの処理速度を向上させるため、高速な SAT solver を式の分解手続きに用いる手法を提案する。

[1]線形時相論理の充足可能性問題ではないことに注意されたい。

1.3 本論文の構成

本論文の構成は次の通りである。2 章では、リアクティブシステムとその動作仕様に関する諸定義、ならびに仕様が満たすべき性質として充足可能性と段階的充足可能性について紹介する。3 章では、タブロー法による充足可能性判定手続きを紹介し、完全な段階的充足可能性判定手続きを与える。4 章では、終端ノードがタブローグラフのノード集合である特殊な二分決定グラフ $\mathrm{BDD}_{\mathrm{PS}}$, $\mathrm{BDD}_{\mathrm{MPS\_r}}$ の定義を与え、これら 2 種類の二分決定グラフを利用する段階的充足可能性判定手続きの実現法について述べる。

5 章では、タブローグラフ構成手続きに対する並列化手法についてのアイディアを述べ、MPI を利用してそのアイディアを実現する実装法について述べる。引き続き 6 章では、タブローグラフ構成手続きの並列化についての考察を行い、分散オブジェクト技術を用いることで手続きの並列化に加え、構成中のタブローグラフを実行環境全体で保持可能とする手続きの実装手法について述べる。さらに 7 章では、4 章で与える二分決定グラフを用いるグラフ決定化手続き、および行き止まり判定手続きについても分散オブジェクト技術を利用する並列分散化実装を与え、段階的充足可能性判定器を並列分散環境上に実現する手法を示す。

8 章では、タブローグラフ構成の内部手続きとして用いる式の分解手続きの実現方法について、SAT solver を利用する実装法を新たに提案する。最後に 9 章で本論文のまとめを述べる。また、本論文における各章の関係は以下の図 1.1 の通りである。

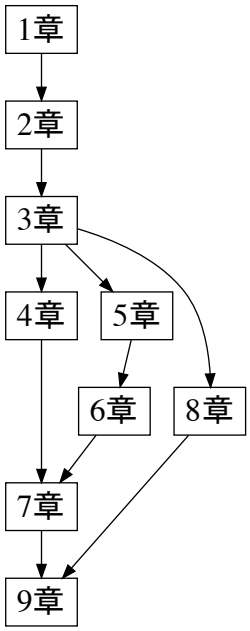

図 1.1: 本論文における各章の関係

第2章　準備

リアクティブシステムは、システムの外部環境からの要求履歴に応じて適切なタイミングで応答するシステムである。特に、本論文では**命題線形時相論理** (*Propositional Linear Temporal Logic, PLTL*) [33] によって動作仕様が形式的に記述されるリアクティブシステムを対象とする。

本章では、リアクティブシステムやその動作仕様、振る舞いを形式的に取り扱うための準備として、それぞれの形式的な定義と、リアクティブシステムの動作仕様が満足すべき性質について紹介する。基本的にこれらの定義は [55] に準じたものを用いる。

2.1　リアクティブシステム

ここでは、リアクティブシステムとその振る舞いについて形式的な定義を与える。

定義 2.1 (リアクティブシステム)
リアクティブシステムは、三つ組 $RS = \langle \mathbb{R}, \mathbb{S}, \mathbf{r} \rangle$ である。三つ組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**の有限集合である。要求イベントとはシステムの外部環境から入力されるイベントのことである。あるタイミングで入力される要求イベントは一般に複数存在し、タイミングごとの要求イベントの集合をタイミング順に並べた列を**要求イベント集合列**と呼ぶ。要求イベント集合列は、要求イベントの集合の有限列もしくは可算無限列である。

- $\mathbb{S}$ は**応答イベント**の有限集合である。応答イベントとはシステムが外部環境への応答として生起させるイベントのことである。要求イベントと同様、システムがあるタイミングで生起させる応答イベントは、一般に複数存在する。タイミングごとの応答イベントの集合

をタイミング順に並べた列を**応答イベント集合列**と呼ぶ。応答イベント集合列は、応答イベントの集合の有限列もしくは可算無限列である。

- $\mathbf{r}$ はリアクティブシステムの動作を規定する関数で**リアクション関数**と呼ぶ。このリアクション関数は、あるタイミングまでの有限長の要求イベント集合列から、そのタイミングでリアクティブシステムが生起すべき集合を決定する関数である。

リアクティブシステムの振る舞いはイベント集合の列で表現される。それぞれのイベント集合は、あるタイミングにおける要求イベントの集合とそのタイミングでの応答イベントの集合の和となる。

定義 2.2 (振る舞い)
リアクティブシステムの**振る舞い**を次のように定義する。ある要求イベント集合列 $\hat{r} = R_0R_1R_2\cdots$ に対するリアクティブシステム $RS = \langle \mathbb{R}, \mathbb{S}, \mathbf{r} \rangle$ の振る舞いを $\mathrm{bhvr}(\mathbf{r}, \hat{r})$ と表し、

$$\mathrm{bhvr}(\mathbf{r}, \hat{r}) = (R_0 \cup S_0)(R_1 \cup S_1)(R_2 \cup S_2)\cdots$$

と定める。ただし $S_i = \mathbf{r}(R_0R_1R_2\cdots R_i),\ (0 \leq i)$ とする。

また、応答イベント集合列 $S_0S_1S_2\cdots$ を $\hat{s}$ と表すとき、表記 $\langle \hat{r}, \hat{s} \rangle$ は、i 番目のイベント集合が $R_i \cup S_i$ となる振る舞いを表すものとする。

たとえば、要求イベント集合列 $\hat{r}$ と、$\hat{r}$ に対する応答イベント集合列 $\hat{s}$ が以下に示す通りであるとき、

$$\hat{r} = \{r_0, r_1\}\{r_0, \neg r_1\}\cdots,\ \hat{s} = \{s_0, \neg s_1\}\{s_0, s_1\}\cdots$$

振る舞いは次のようになる。

$$\mathrm{bhvr}(\mathbf{r}, \hat{r}) = \langle \hat{r}, \hat{s} \rangle = \{r_0, r_1, s_0, \neg s_1\}\{r_0, \neg r_1, s_0, s_1\}\cdots$$

2.2 リアクティブシステム仕様

本論文では、リアクティブシステムの要求イベントや応答イベントの生起を命題変数に対応させ、その上での命題線形時相論理（PLTL）をリ

アクティブシステム仕様の記述言語とする。したがって、リアクティブシステムの動作は PLTL によって記述された論理式により決定される。

本論文で用いる PLTL は、様相演算子として “**weak until** 演算子 $[\cdot]$” を含む命題線形時相論理である。演算子 “$[\cdot]$” と、一般に用いられる weak until 演算子 “$\mathcal{W}$” との関係は $[g]f \equiv f\mathcal{W}g$ である。

定義 2.3 (論理式)
$\mathcal{P}$ を命題変数の集合とするとき、$\mathcal{P}$ を基に構成される通常の命題論理式の定義に以下の規則を加える。f, g が式であるとき、$[g]f$ は式である。$\langle g\rangle f$, $\Box f$, $\Diamond f$ は、それぞれ $\neg[g]\neg f$, $[\bot]f$, $\langle\bot\rangle f$ の略記とする。また、様相演算子を持つ $[g]f, \neg[g]f$ の形をした論理式を**時間式**と呼ぶ。

定義 2.4 (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。イベント集合列 σ の i 番目のイベント集合が $\mathcal{P}$ 上の式 f を満たすことを $\langle\sigma, i\rangle \models f$ と表し、次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)。
- $\langle\sigma, i\rangle \models [g]f \iff (\forall j \geq 0)\ \langle\sigma, i+j\rangle \models f$ または $(\exists j \geq 0)\big(\langle\sigma, i+j\rangle \models g$ かつ $\forall k(0 \leq k < j)\ \langle\sigma, i+k\rangle \models f\big)$。

その他の論理的結合子の意味は、通常の命題論理式と同様である。以下、$\langle\sigma, 0\rangle \models f$ を単に $\sigma \models f$ と表記する。

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 $\langle\mathcal{R}, \mathcal{S}, \varphi\rangle$ と定める。三つ組の各要素は以下の通りである。

- $\mathcal{R}$ 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。

たとえば、要求イベントと応答イベントをそれぞれ一つずつ持つリアクティブシステムの動作仕様として、“システムは‘要求イベントが入力されたら、いつか応答イベントを生起させる’ ことを常に満たし続ける”という簡単なものを考える。各イベントに対応する要求命題、応答命題をそれぞれ r, s とすると、その動作仕様は $\langle \{r\}, \{s\}, \Box(r \to \Diamond s) \rangle$ と表される。

2.3 動作仕様が満足すべき性質

本節では、実現可能なリアクティブシステム仕様が満たすべき性質のうち、充足可能性と段階的充足可能性 [55] について扱う。

実現可能なリアクティブシステム仕様が満たすべき性質の一つとして、充足可能性（無矛盾性）がよく知られている。“リアクティブシステム仕様が充足可能である” とは、直観的には “仕様を表現した論理式に矛盾せずに、動作し続けられる振る舞いが存在する” ことを表す。形式的な定義は次の通りである。以下では、集合 A の要素に対し、その可算無限長の列をすべて集めた集合を A^ω と表す。

定義 2.6 (充足可能性)
リアクティブシステムの動作仕様 $\langle \mathcal{R}, \mathcal{S}, \varphi \rangle$ 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。

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

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、$\left(2^{\mathbb{R}}\right)^\omega$、$\left(2^{\mathbb{S}}\right)^\omega$ はそれぞれ、可算無限長の要求イベント集合列の集合、可算無限長の応答イベント集合列の集合である。

リアクティブシステムが実際に稼働する場合を考えると、ある要求イベント集合列に対して、仕様を満たしながら動作し続けることのできる応答イベント集合列が存在した（充足可能性を満たしている）としても、実際にシステムがその振る舞いをたどるように各時刻で応答イベント集合を決定できるとは限らない。実際に稼働するシステムでは、ある時刻での応答イベント集合はその時刻までのシステムの振る舞いと、その時刻における要求イベント集合のみで決定しなければならず、将来充足性を満たせなくなる応答イベント集合を選んでしまう可能性がある。

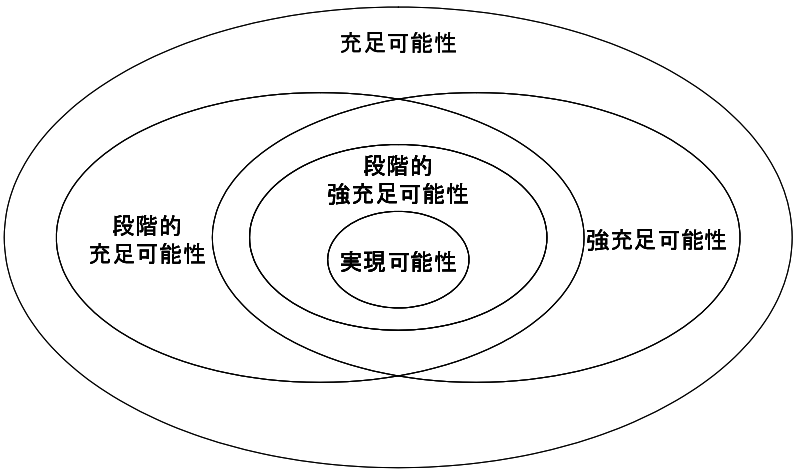

図 2.1: 動作仕様の満たすべき性質の包含関係

仕様が充足可能性を満たしていたとしても、将来の充足性を維持するために未来を予測した応答イベント集合の決定が必要となる動作仕様は、やはり仕様として不適切である。システムが実現可能であるためには未来を予測することなく実行の各段階で仕様を満たすように応答する方法が存在しなければならない。この性質を仕様の**段階的充足可能性**とよぶ。以下では、集合 A の要素に対し、その有限長の列をすべて集めた集合を A^+ と表す。

定義 2.7 (段階的充足可能性 [55])
リアクティブシステムの動作仕様 $\langle \mathcal{R}, \mathcal{S}, \varphi \rangle$ 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。

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

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。また、$\mathrm{bhvr}(\mathbf{r}, \bar{r}) \cdot \langle \tilde{r}', \tilde{s}' \rangle$ は、有限長の振る舞い $\mathrm{bhvr}(\mathbf{r}, \bar{r})$ の後ろに可算無限長のイベント集合列 $\langle \tilde{r}', \tilde{s}' \rangle$ が続くイベント集合列を表す。

また、[30] では、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、**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。これらの性質も実現可能性の必要条件で、仕様が実現可能性を失った原因を特定することを目的として提案された。それぞ

れの性質の包含関係は図 2.1 の様になる。本論文においては、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。

強充足可能性　任意の無限要求イベント集合列に対して、動作仕様を満たす応答列が存在する。形式的な定義は以下の通りである。

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

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。形式的な定義は以下の通りである。

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

第3章　タブロー法による性質判定手続き

本章では、タブロー法を用いた動作仕様の充足可能性判定手続きと段階的充足可能性判定手続きについて述べる。これらは、5 章、6 章で提案する性質判定手続きの並列化実装に密接に関わるため、その実現方法の詳細まで紹介する。

3.1　充足可能性判定

PLTL によって記述された動作仕様の充足可能性を判定する手続きとして、タブロー法を用いた手続きが [26][55] にて提案されている。また、この手続きを基にいくつかの効率化手法を導入した充足可能性判定器の実現方法が [52] にて提案されている。本節では [52] にて提案されている実現方法について紹介する。

充足可能性の定義から仕様が充足可能であるとは、仕様を満足したまま動作し続けることのできる無限パスを少なくとも一つ持つ状態遷移モデルが構成できることと同義である。したがって充足可能性の判定は、仕様からそういった状態遷移モデルが構成できるかを確認すればよい。

タブロー法による充足可能性判定手続きでは、仕様から、その仕様を満たす状態遷移モデルをすべて含むグラフを構成する。このグラフを**タブローグラフ**と呼ぶ。仕様から構成されたタブローグラフ中に、初期ノードから仕様を満足したままで遷移し続けられる無限パスが存在すれば、仕様は充足可能であると判定される。

3.1.1　プレステートグラフ表現

本論文では、タブローグラフとして青島らによって提案された**プレステートグラフ** [52] を用いる。以下では、プレステートグラフの形式定義、な

らびに仕様からプレステートグラフを構成する手続きについて紹介する。

定義 3.1 (プレステートグラフ [52])
プレステートグラフは三つ組 $\langle V_p, v_{p0}, E_p \rangle$ である。プレステートグラフのノードを**プレステート**と呼ぶ。V_p はプレステートの集合で、初期プレステートを除く各プレステートは時間式の集合である。v_{p0} は初期プレステート、$E_p \subseteq V_p \times V_p$ はエッジの集合である。

PLTL 式によって記述された仕様から構成されるプレステートグラフでは、初期プレステートは仕様の式そのものであり、各プレステートは次の時刻にまで持ち越される制約の集合となっている。

次に示す**式の分解手続き**は、プレステート中の各論理式を分解し、プレステートが表す制約を満足するために次の時刻で生起すべきイベントと、次プレステート、つまり、さらに一つ先の時刻にまで持ち越される制約を計算する手続きである。一般に、次の時刻で成立すべきイベントと次プレステートの組み合わせは複数存在する。

手続き 3.1 (式の分解手続き [55])
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 $decomp(S)$ 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 を集合 $\big(S_i - \{f_{ij}\}\big) \cup \{f\}$ に置き換える。
   (b) f_{ij} が $f_1 \wedge f_2$ という形の式ならば、S_i を集合 $\big(S_i - \{f_{ij}\}\big) \cup \{f_1, f_2\}$ に置き換える。
   (c) f_{ij} が $\neg(f_1 \wedge f_2)$ という形の式ならば、S_i を 2 つの集合 $\big(S_i - \{f_{ij}\}\big) \cup \{\neg f_1\}$, $\big(S_i - \{f_{ij}\}\big) \cup \{\neg f_2\}$ に置き換える。
   (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 $\big(S_i - \{f_{ij}\}\big) \cup \{f_2\}$, $\big(S_i - \{f_{ij}\}\big) \cup \{f_1, \neg f_2, [f_2]f_1\}$ に置き換える。

(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 $\big(S_i - \{f_{ij}\}\big) \cup \{\neg f_1, \neg f_2\}$, $\big(S_i - \{f_{ij}\}\big) \cup \{f_1, \neg f_2, \neg[f_2]f_1)\}$ に置き換える。

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。

式の分解手続きを繰り返し用いることで、PLTL によって記述された仕様からプレステートグラフを構成することができる。プレステートに式の分解手続きを適用し得られる集合の各要素を**ステート**とよび、ステートから時間式だけを取り出し作った時間式の集合が、次プレステートとなる。そのため、プレステートから次プレステートへの遷移にはラベルとしてステートが付与されていると見なすことができる。さらに言えば、ステートの含んでいる生起すべきイベント集合を遷移のラベルと見なすことができる。

定義 3.2 (時間式集合)
S に含まれる時間式の集合を $temp(S)$ と表す。また、式集合の集合 Σ について、時間式集合の集合 $\{temp(S) \mid S \in \Sigma\}$ を $Temp(\Sigma)$ と表記する。

手続き 3.2 (プレステートグラフ構成手続き [52])
式 φ を入力とし、プレステートグラフ $\langle V_p, v_{p0}, E_p \rangle$ を出力する。

```
1:  constPSGraph(φ : formula)
2:     v_p0 := {φ}, V_p := {v_p0}, E_p := ∅, V'_p := ∅;
3:     while (p ∈ V_p − V'_p)
4:        P := Temp(decomp(p));
5:        V_p := V_p ∪ P;
6:        while (p' ∈ P)
7:           E_p := E_p ∪ {⟨p, p'⟩};
8:        V'_p := V'_p ∪ {p};
```

2–3 行目: V'_P は分解済みのノードの集合を表し、p はまだ分解されていないノードを表す。

また、[52] では上記の手続きにおいて、プレステートに対する式の分解手続き演算を BDD を利用して実装する方法、並びに BDD で表された式集合の集合 Σ から $Temp(\Sigma)$ を計算する手法を与えている。式の分解手続

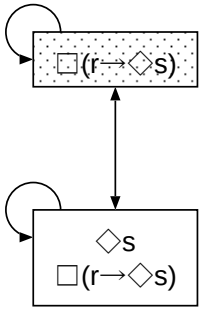

図 3.1: プレステートグラフ

きの結果を保持しておくことで、同じ式に対する処理を何度も行う必要がなくなる。また、結果を BDD を用いて表すことで分解手続きの結果保持に利用する空間的なコストを抑えている。

図 3.1 は、式 $\Box(r \rightarrow \Diamond s)$ に対して手続き 3.2 を適用して構成したプレステートグラフである。この図では、初期プレステートを網掛けのかかったノードで表している。

3.1.2 イベンチャリティ制約の確認

$\neg[g]f$ の形をした論理式は、“式 g が成立する前のある時点で、式 $\neg f$ が成立する” ことを表現している。このような “いつか成立する” という論理式を**イベンチャリティ式**と呼ぶ。イベンチャリティ式を含む仕様が充足可能であるためには、この “いつか成立する” という**イベンチャリティ制約**がすべて満足されなければならない。

プレステートグラフを用いた充足可能性判定手続きの場合、あるイベンチャリティ制約 $\neg[g]f$ を満足するかどうかの確認は、この式を含むプレステートから、この式を含まないプレステートに到達可能であるか確認することで行う。

この確認を効率的に行うための手法として、[52] ではプレステートグラフを極大強連結成分に分割し、その極大強連結成分の中に次で定義する**自己充足**な極大強連結成分が存在するか確認することで行うという手法が提案されている。

定義 3.3 (自己充足)
次の 2 つの条件を満たす部分プレステートグラフ G を**自己充足**であると

呼ぶ。

- G 中に少なくとも一つのエッジを持つ。
- G に含まれるプレステート p 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 p から到達可能な $\neg[g]f$ を含まないプレステートが存在する。

自己充足の定義に含まれる “少なくとも一つのエッジを持つ” との項目は、“自己充足な強連結成分” という表現にエッジを持たない一つのプレステートのみで構成される強連結成分を排除するための項目である。これは、自己充足な強連結成分が無限長の振る舞いが存在していることを表しているのに対し、エッジを持たない強連結成分は無限長の振る舞いを表さず、充足可能の証拠にはなり得ないためである。

手続き 3.2 を用いて仕様から構成したプレステートグラフに、自己充足な極大強連結成分が含まれていた場合、その仕様は充足可能であると判定される。逆に、自己充足な極大強連結成分が含まれていない場合、その仕様は充足不能である。

3.1.3 充足可能性判定手続き

3.1.1 節、 3.1.2 節で紹介したプレステートグラフ構成手続き、およびイベンチャリティ制約の確認手法を用いて仕様の充足可能性判定手続きは、次のように実現できる。

手続き 3.3 (充足可能性判定手続き)
仕様を表す式 φ を入力とし、式 φ が充足可能であるかを判定する。

1. 式 φ に手続き 3.2 を適用しプレステートグラフを構成する。
2. プレステートグラフを極大強連結成分に分割し、それらの中に自己充足な極大強連結成分が含まれていれば、充足可能と、含まれていなければ充足不能と判定する。

仕様を満たす “無限長の要求イベント集合列とそれに対応する無限長の応答イベント集合列の組（無限長の振る舞い）” が一つでも存在すればその仕様は充足可能である。仕様から構成されたプレステートグラフにお

いて初期ノードからのパスは、リアクティブシステムの振る舞いを表しており、プレステートグラフ中に自己充足可能な強連結成分が含まれることは、イベンチャリティ制約を解消できる無限長の振る舞いの存在を意味する。

3.2 段階的充足可能性判定

充足可能性判定手続きと同様に、PLTL によって記述された動作仕様の段階的充足可能性判定手続きとして、タブロー法を用いた手続きが [55] にて提案されている。また、この判定手続きを基にした判定手続きの実装法が [52] にて示されている。

しかし、このどちらの判定手続きも段階的充足可能な仕様を、段階的充足不能と判定してしまう恐れのある手続きであり、後に [44] によって完全な手続きが示され、オートマトンを用いた完全な判定法が [53] にて提案されている。

本節では、[53] にて示された決定性オートマトンを用いた判定法を、タブロー法に応用した段階的充足可能性の完全な判定手続きについて述べる。

タブロー法を用いた仕様の段階的充足可能性判定では、仕様からシステムの振る舞いに関して決定的な状態遷移系を表すグラフを作り、そのグラフの各ノードがどんな入力イベント集合に対しても対応する遷移が存在することを確認しなければならない。

本論文で提案する段階的充足可能性判定手続きは、次の四つの部分手続きを順に行い、最後の行き止まり判定手続きの結果から、仕様が段階的充足可能性を満たすかどうかを判定する。

1. プレステートグラフ構成手続き（手続き 3.2）
2. イベンチャリティ制約違反ノード除去手続き
3. イベント集合によるグラフの決定化手続き
4. 行き止まり判定手続き

以降では、3.1.1 節で紹介したプレステートグラフ構成手続きを除く部分手続きについて述べる。

3.2.1　イベンチャリティ制約違反ノード除去手続き

この手続きでは、プレステートグラフ中に現れるイベンチャリティ制約を満足しないノードをすべて削除する。イベンチャリティ違反ノードがすべて除去されたプレステートグラフは、システムがグラフ中のどのプレステートで表現される状態にあったとしても、その状態で充足性を満足するという特徴を持つ。

本手続きは次の手順で、プレステートグラフからイベンチャリティ制約を違反しているすべてノードを削除する。

1. プレステートグラフを極大強連結成分に分解する。
2. 以下をプレステートグラフが変化しなくなるまで繰り返す。
   (a) 各極大強連結成分について、その極大強連結成分が自己充足であるか、もしくはその極大強連結成分中のノードから自己充足な極大強連結成分中のノードに到達可能であるかを確認する。
   (b) そのどちらでもない極大強連結成分に含まれるノードをすべて削除する。

この手続きで、プレステートグラフのルートノードが削除された場合、仕様は充足不能である。充足可能性は段階的充足可能性の必要条件であるため、段階的充足可能性判定において、この部分手続きでプレステートグラフのルートノードが削除された場合は、この時点で段階的充足不能であると判定する。

3.2.2　イベント集合によるグラフの決定化手続き

一般に、イベンチャリティ制約違反ノード除去手続きを適用した後のプレステートグラフには、ある振る舞いを表現するグラフのたどり方が複数存在する。この手続きは、プレステートグラフをリアクティブシステムの振る舞いによって決定化し、振る舞いとグラフのたどり方が1対1に対応するグラフを構成する手続きである。プレステートグラフを決定化して得られるグラフを**マクロプレステートグラフ**と呼び、マクロプレステートグラフのノードを**マクロプレステート**と呼ぶ。また以下では、この手続きを**マクロプレステートグラフ構成手続き**とも呼ぶ。

マクロプレステートグラフの形式的な定義とグラフの決定化手続きを以下に示す。

定義 3.4 (マクロプレステートグラフ)
プレステートグラフ $\langle V_p, v_{p0}, E_p \rangle$ を要求、応答イベント集合で決定化したグラフを**マクロプレステートグラフ**と呼び、そのノードを**マクロプレステート**と呼ぶ。マクロプレステートはプレステートの集合である。マクロプレステートグラフは三つ組 $\langle V_m, v_{m0}, E_m \rangle$ で表し、$V_m \subseteq 2^{V_p}$, $v_{m0} = \{v_{p0}\} \in V_m$, $E_m \subseteq V_m \times V_m$ である。

手続き 3.4 (グラフの決定化手続き)
プレステートグラフ $\langle V_p, v_{p0}, E_p \rangle$ を入力とし、マクロプレステートグラフ $\langle V_m, v_{m0}, E_m \rangle$ を出力する。

1: *determinization*($\langle V_p, v_{p0}, E_p \rangle$: Prestate Graph)
2: $v_{m0} := \{v_{p0}\},\ V_m := \{v_{m0}\},\ V'_m := \emptyset,\ E_m := \emptyset$;
3: **while** $(m \in V_m - V'_m)$
4: **while** $(s \in 2^{\mathcal{R} \cup \mathcal{S}})$
5: $m' := \bigcup_{p \in m} \{temp(u) \mid u \in decomp(p) \ \wedge \ u \supseteq s\}$
6: $V_m := V_m \cup \{m'\}$;
7: $E_m := E_m \cup \{\langle m, m' \rangle\}$;
8: $V'_m := V'_m \cup \{m\}$;

4 行目: $\mathcal{R}, \mathcal{S}$ は、それぞれプレステートグラフ中に現れる要求命題集合、応答命題集合を表す。
5–7 行目: m' は m から直接到達可能なマクロプレステートで、m 中の各プレステートからステート s をラベルに持つ遷移で直接到達可能なプレステートをすべて集めたプレステート集合である。

3.2.3 行き止まり判定手続き

ある要求イベント集合に対して対応する遷移が存在しないマクロプレステートグラフ中のノードを**行き止まり**のマクロプレステートと呼ぶ。行き止まり判定手続きは、グラフが変化しなくなるまでマクロプレステートグラフから行き止まりのマクロプレステートの除去を繰り返し、マクロプレステートグラフの初期ノードが行き止まりと判定されるかどうかを確認する手続きである。

マクロプレステートグラフ中に行き止まりが存在しない場合、どんな有限長のイベント集合列に対しても、対応する振る舞いを表す遷移をそのグラフ中をたどる遷移の中から見つけることができる。したがって、動作仕様から行き止まりのないマクロプレステートグラフが構成できれば、そのグラフ中の遷移をたどる限りどんな有限長のイベント集合列に対しても仕様を満足したままで振る舞うことができ、その後も充足性を満足するという性質、つまり段階的充足可能性が満たされることになる。

逆に、動作仕様から構成したマクロプレステートグラフの初期ノードが行き止まりと判定されて除去されてしまう場合は、段階的充足可能性を満足できないことを表す。

したがって、段階的充足可能性判定では、この行き止まり判定手続きにおいて初期マクロプレステートが行き止まりと判定された場合には段階的充足不能であると、初期マクロプレステートが行き止まりではないと判定された場合には段階的充足可能であると判定する。

以下に、行き止まりの形式的定義とマクロプレステートグラフの行き止まり判定手続きを示す。

定義 3.5 (行き止まり)
マクロプレステートが、ある要求イベント集合で遷移できる次ノードを持たない場合、そのマクロプレステートは**行き止まり**であると呼ぶ。

手続き 3.5 (グラフの行き止まり判定手続き)
マクロプレステートグラフを入力とし、そのマクロプレステートグラフの初期マクロプレステートが行き止まりであれば真を、そうでなければ偽を出力する。本手続きは、以下のステップで処理される。

1. 以下をグラフが変化しなくなるまで繰り返す。
   (a) すべてのマクロプレステートについてそのマクロプレステートが行き止まりであるか判定。
   (b) 行き止まりと判定されたノードすべてを、マクロプレステートグラフ中から除去。
2. グラフ中に初期マクロプレステートが残っていなければ真を、そうでなければ偽を出力する。

3.2.4 段階的充足可能性判定手続き

これまでに述べた部分手続きをあわせることで、段階的充足可能性判定手続きを以下のように構成できる。

手続き 3.6 (段階的充足可能性判定手続き)
動作仕様の段階的充足可能性は以下のステップで判定できる。

1. 動作仕様からプレステートグラフを構成する。
2. ステップ 1 で構成されたプレステートグラフ中に含まれる、すべてのイベンチャリティ違反ノードをグラフから除去する。初期ノードが除去された場合、偽を出力して終了。
3. ステップ 2 で変更されたプレステートグラフをもとに手続き 3.4 を用いて要求、応答イベント集合でグラフを決定化する。
4. マクロプレステートグラフについて手続き 3.5 を用いて要求イベントに対する行き止まり判定をおこない、決定化グラフの初期ノードが行き止まりと判定されれば、偽を出力。そうでなければ真を出力する。

イベンチャリティ制約に違反するノードをすべて取り除いたプレステートグラフでは、どのプレステートからも自己充足な強連結成分へのパスが存在するため、初期ノードからそのプレステートへ到達するどんな有限長の振る舞いも、充足性を保存することとなる。

仕様が段階的充足可能であるためには、リアクティブシステムは任意の有限長の要求イベント集合列に対して振る舞った後にも充足性を保存していなければならない。つまり、任意の有限長の要求イベント集合列に対する振る舞いが、イベンチャリティ制約に違反するノードをすべて取り除いたプレステートグラフ中の有限長のパスとして表現できるか確認する必要がある。しかし、プレステートグラフはイベント集合に対して決定化されておらず、任意の有限長の振る舞いについて、プレステートグラフ中に対応するパスが存在するか確認することは困難である。

プレステートグラフをイベント集合によって決定化しマクロプレステートグラフを構成すると、マクロプレステートグラフ上の有限長のパスとリアクティブシステムの有限長の振る舞いが 1 対 1 に対応する。

マクロプレステートグラフ中に現れる行き止まりのノードは、そのノードへ到達する有限長の振る舞いの後に、さらに次の要求イベント集合に対し応答しようとしたときに、充足性を満足したままで応答できない要求イベント集合が存在することを表す。そのような行き止まりのノードをマクロプレステートグラフからすべて取り除いても、グラフが残っている場合、任意の有限長の要求イベント集合列に対してそのマクロプレステートグラフ上のパスとして表現できるリアクティブシステムの振る舞いが存在し、その振る舞いの後にも充足性を保存していることを意味する。つまり、マクロプレステートグラフからすべての行き止まりを取り除いたグラフが段階的充足可能の証拠となる。

第4章　二分決定グラフを利用した段階的充足可能性判定手続きの実装

本章では、前章の段階的充足可能性判定手続きの二つの内部手続き、グラフの決定化手続きと行き止まり判定手続きついての実装手法を提案する。ここでは、2 種類の二分決定グラフ $\mathrm{BDD_{PS}}$, $\mathrm{BDD_{MPS\_r}}$ を導入し、グラフの決定化手続きの実装では $\mathrm{BDD_{PS}}$、行き止まり判定手続きの実装では $\mathrm{BDD_{MPS\_r}}$ を用いる実装手法を示す。

4.1　プレステートグラフの決定化手続きの実装

プレステートグラフを決定化する際、構成されるマクロプレステートグラフ中のノード（マクロプレステート）間の遷移を計算する必要がある。あるマクロプレステートの次ノード集合を計算するためには、そのマクロプレステートを構成する各プレステートがどんなイベント集合によってどのプレステートへ遷移するかという情報が必要になる。

例えば、プレステート p_1, p_2 を用いて $m = \{p_1, p_2\}$ と表されるマクロプレステート m について、あるイベント集合 s に対する次ノード m' を計算することを考える。手続き 3.4 に示した通り、この計算にはイベント集合 s に対するプレステート p_1, p_2 の次ノードがどのプレステートであるか知る必要がある。それぞれのプレステートの s に対する次ノードを p'_1, p'_2 とすると、$m' = \{p'_1, p'_2\}$ と計算されることとなる。これを任意のイベント集合に対して計算することで、マクロプレステート m のすべての次ノードが計算されることとなる。

本節では、各プレステートについての “どんなイベント集合によってどのプレステートへ遷移するか” という情報を表現するデータ構造として、二分決定グラフ $\mathrm{BDD_{PS}}$ を導入する。そして、$\mathrm{BDD_{PS}}$ 同士の和集合演算

を定義し、その演算を利用することで、構成中のマクロプレステートの次ノード計算を行う決定化手続きの実装法を与える。上述の例でたとえると、本節で与える実装は、プレステート p_1, p_2 に対する $\mathrm{BDD_{PS}}$ 同士の和集合演算により、マクロプレステート m の、m' を含めたすべての次ノード集合を計算する実装となっている。

4.1.1 $\mathbf{BDD_{PS}}$

まず、二分決定グラフ $\mathrm{BDD_{PS}}$ について形式的な定義と、その性質について述べる。

定義 4.1 ($\mathbf{BDD_{PS}}$)
BDD_{PS} は、一般的な BDD 同様、二分決定グラフである。非終端ノードは、要求変数、および応答変数によりラベル付けされ、終端ノードはプレステートの集合によりラベル付けされる。根ノードから終端ノードへのパス中に現れる非終端ノードの変数順序は固定されており、任意の要求命題はどの応答命題よりも先に現れると定める。また、$\mathrm{BDD_{PS}}$ は ROBDD と同様の規則によって簡約化でき、以降では、すべての $\mathrm{BDD_{PS}}$ は簡約化されているものとする。

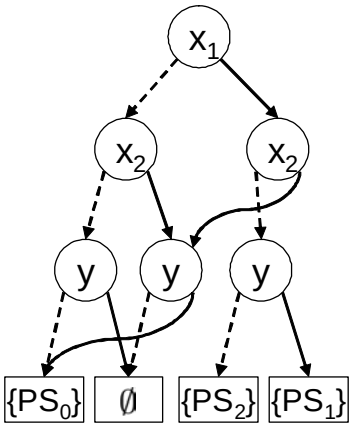

図 4.1: $\mathrm{BDD_{PS}}$

ある $\mathrm{BDD_{PS}}$ の根ノードから終端ノードへのパスは、そのパスに対応する論理式集合と、終端ノードにラベル付けされたプレステート集合とを対応付ける関係を表す。図 4.1 の例において、実線を high エッジ、破線

を low エッジとすると、論理式集合 $\{x_1, \neg x_2, y\}$, $\{\neg x_1, x_2, \neg y\}$ は、プレステート集合 $\{\mathrm{PS}_1\}, \emptyset$ にそれぞれ対応づけられる。

プレステートグラフ中の各プレステートについて、イベント集合とそれに対応する次プレステートを $\mathrm{BDD_{PS}}$ で表現するとき、根ノードから終端ノードへの各パスがイベント集合を表す。あるパスをたどって到達する終端ノードは、そのパスが表すイベント集合での次プレステートのみが含まれるプレステート集合がラベル付けされる。そのプレステートに、イベント集合に対する次プレステートが存在しない場合は、そのイベント集合に対応するパスをたどった終端ノードは空集合がラベル付けされる。

次に、二つの $\mathrm{BDD_{PS}}$ の和集合を表す $\mathrm{BDD_{PS}}$ を次のように定める。

定義 4.2 ($\mathrm{BDD_{PS}}$ の和集合)
二つの $\mathrm{BDD_{PS}}$ の和集合を表す $\mathrm{BDD_{PS}}$ とは、各論理式集合で到達できる終端ノードが、元の二つの $\mathrm{BDD_{PS}}$ において同じ論理式集合で到達できる終端ノードの和集合になっている $\mathrm{BDD_{PS}}$ のことである。たとえば、図 4.2 に示す二つの $\mathrm{BDD_{PS}}$ の和集合は、図 4.3 に示した $\mathrm{BDD_{PS}}$ の様になる。

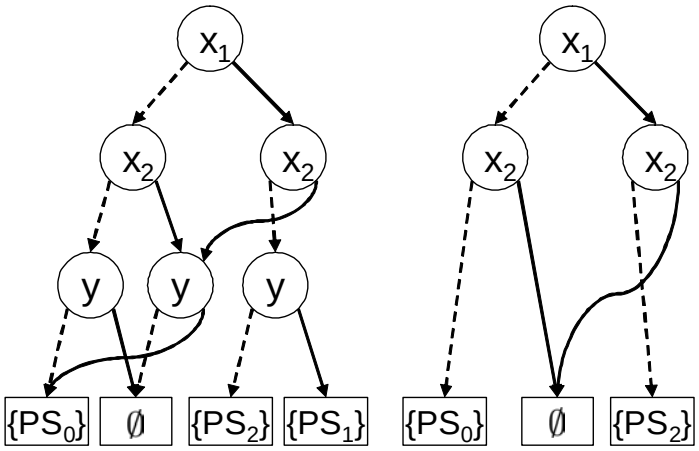

図 4.2: 和集合をとる 2 つの $\mathrm{BDD_{PS}}$

$\mathrm{BDD_{PS}}$ の和集合演算を用いることにより、プレステートの遷移先を表現する $\mathrm{BDD_{PS}}$ 同士を和集合演算で合成できる。その合成結果を表す $\mathrm{BDD_{PS}}$ は “プレステート集合がイベント集合の生起で、どういったプレステート集合に遷移するのか” を表現している。$\mathrm{BDD_{PS}}$ の和集合演算のこの特徴を利用することで、プレステートグラフの決定化手続きが実装できる。

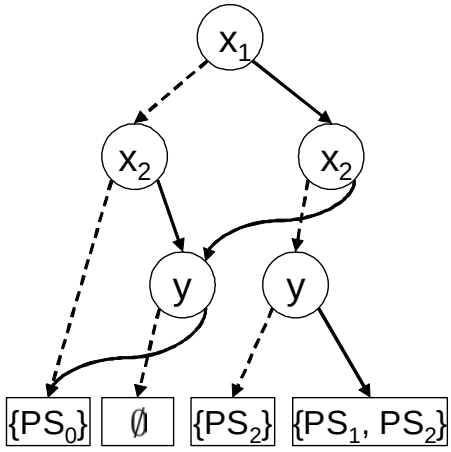

図 4.3: 和集合の結果を表す BDD_{PS}

4.1.2 BDD_{PS} を利用した決定化手続きの実装

マクロプレステートはプレステートの集合として表され、マクロプレステートのあるイベント集合に対する次ノードは、マクロプレステート中に含まれる各プレステートについてのそのイベント集合に対する次プレステートの集合として計算される。

本節で提案する実装法では、このマクロプレステートの次ノードの計算に BDD_{PS} の和集合演算を用いる。マクロプレステートを構成するプレステートから計算された BDD_{PS} を和集合演算で合成する。その結果得られた BDD_{PS} の終端ノードは、マクロプレステート中の各プレステートから、あるイベント集合によって遷移できるプレステートの集合を表している。すなわち、BDD_{PS} の和集合演算により得られた BDD_{PS} は、そのマクロプレステートについてのイベント集合の生起とそのイベント集合での遷移先のマクロプレステートを表現していることになる。

例えば、あるマクロプレステート MPS が $MPS = \{PS_0, PS_1\}$ と表され、図 4.2 の左側の BDD_{PS} をプレステート PS_0 の BDD_{PS}、図 4.2 の右側の BDD_{PS} をプレステート PS_1 の BDD_{PS} とする。このとき、PS_0 と PS_1 の BDD_{PS} 同士の和集合結果は図 4.3 の様になり、この結果はマクロプレステート MPS の次ノードが、$\{PS_0\}$, $\{PS_1, PS_2\}$, $\{PS_2\}$ の三つであることを示している。

以下に BDD_{PS} とこの和集合演算を利用したプレステートグラフの決定化手続きを示す。

手続き 4.1 ($\mathbf{BDD_{PS}}$ を利用するグラフ決定化)
プレステートグラフ $\langle V_p, v_{p0}, E_p \rangle$ を入力とし、マクロプレステートグラフ $\langle V_m, v_{m0}, E_m \rangle$ を出力する。ただし、以下の擬似コード中に現れる *convBddPs* は引数の論理式集合を対応する $\mathrm{BDD_{PS}}$ に変換する関数、*getLeaves* は引数の $\mathrm{BDD_{PS}}$ の終端ノードのラベルをすべて集め、集合 (つまり、プレステート集合の集合) として取り出す関数とする。

1: *determinization*($\langle V_p, v_{p0}, E_p \rangle$: Prestate Graph)
2: $v_{m0} := \{v_{p0}\},\ V_m := \{v_{m0}\},$
$V'_m := \emptyset,\ E_m := \emptyset$;
3: **while** $(m \in V_m - V'_m)$
4: $u := \bigcup_{p \in m} convBddPs(decomp(p))$;
5: $M := getLeaves\,(u)$;
6: $V_m := V_m \cup M$;
7: **while** $(m' \in M)$
8: $E_m := E_m \cup \{\langle m, m' \rangle\}$;
9: $V'_m := V'_m \cup \{m\}$;

4.2 マクロプレステートグラフの行き止まり判定手続きの実装

段階的充足可能性判定手続きでは、マクロプレステートグラフに対して行き止まり判定手続きを適用し、初期マクロプレステートが行き止まりと判定されるか確認する必要がある。

行き止まり判定手続きでは、グラフ中のすべてのマクロプレステートについて行き止まりであるか確認する。本節では、マクロプレステートが行き止まりであるかの判定に新たに導入する二分決定グラフ $\mathrm{BDD_{MPS\_r}}$ を用いる行き止まり判定手続きの実装法を示す。

本節で導入する $\mathrm{BDD_{MPS\_r}}$ は、マクロプレステートグラフ中の各マクロプレステートについての遷移情報を表現するために用いる二分決定グラフで表現されたデータ構造で、このグラフの形状を確認することでそのマクロプレステートが行き止まりであるかを判定する。

4.2.1 $\mathbf{BDD_{MPS\_r}}$

二分決定グラフ BDD_{MPS_r} についての形式的な定義は以下の通りである。

定義 4.3 ($\mathbf{BDD_{MPS\_r}}$)
BDD_{MPS_r} は、一般的な BDD 同様、二分決定グラフである。非終端ノードは、要求命題によりラベル付けされ、終端ノードはマクロプレステート集合によりラベル付けされる。また、BDD_{MPS_r} は ROBDD 同様の規則によって簡約化でき、以降では、すべての BDD_{MPS_r} は簡約化されているものとする。

4.1.1 節で述べた BDD_{PS} が、プレステートグラフ中のノードについての遷移情報を表現するために用いられるのに対して、本節で導入する BDD_{MPS_r} は、マクロプレステートグラフ中のノードについての遷移情報を表現するために用いられる。

あるマクロプレステートに対する BDD_{MPS_r} は、システム外部からの要求イベント集合と、その要求イベント集合にシステムが応答する際に遷移する次マクロプレステートの候補集合との対応を表している。この対応関係は、BDD_{MPS_r} の根ノードから終端ノードへのパスが要求イベント集合として表現され、そのパスに対する終端ノードが、その要求イベント集合に対する対応する次ノード集合の候補として表現される。

マクロプレステートが行き止まりであるかを判定するために用いる BDD_{MPS_r} は、決定化手続きでそのマクロプレステートの次ノードを計算するために利用した BDD_{PS} を以下の手続きで変形した得られる。

手続き 4.2 ($\mathbf{BDD_{MPS\_r}}$ への変形)

1. BDD_{PS} の根ノードから終端ノードへ至る各パスについて、根ノードからそのパスをたどった時に初めて出会う応答命題がラベル付けされたノードを探索する。ただし、応答命題がラベル付けされたノードを通らずに終端ノードに到達した場合は、そのパスに関しては以下のステップを行わず、新規の終端ノードを作りそのパスの終端ノードとする。新規の終端ノードのラベルは、元の終端ノードにラベル付けされているマクロプレステートのみを含むマクロプレステート集合とする。

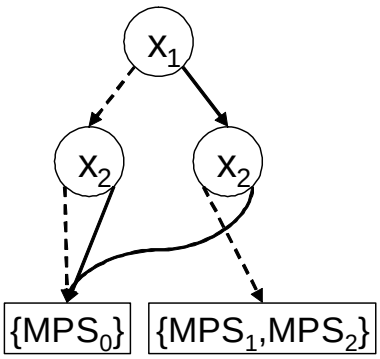

図 4.4: 図 4.3 の $\mathrm{BDD_{PS}}$ から変形した $\mathrm{BDD_{MPS\_r}}$

2. 応答命題がラベル付けされたノードそれぞれについて、そのノードから $\mathrm{BDD_{PS}}$ のパスをたどって到達できる終端ノードにラベル付けされたマクロプレステートをすべて集めた集合を作る。

3. ステップ 2 で作ったマクロプレステート集合をラベルにもつ新たな終端ノードを作り、これをステップ 1 の応答命題がラベル付けされたノードと置き換える。

4.1.1 節で図 4.3 に示した $\mathrm{BDD_{PS}}$ を上で与えた変形法にしたがって $\mathrm{BDD_{MPS\_r}}$ に変形すると図 4.4 のようになる。ただし、プレステート集合 (つまり、マクロプレステート) $\{\mathrm{PS}_0\}$, $\{\mathrm{PS}_1, \mathrm{PS}_2\}$, $\{\mathrm{PS}_2\}$ を、それぞれ MPS_0, MPS_1, MPS_2 と表す。

4.2.2 $\mathrm{BDD_{MPS\_r}}$ を利用した行き止まり判定手続きの実装

あるマクロプレステートに対して $\mathrm{BDD_{MPS\_r}}$ を作ると、その $\mathrm{BDD_{MPS\_r}}$ の形状から、そのマクロプレステートがある要求イベント集合についての次ノードを持つかどうかを判定できる。

上でも述べた通り、$\mathrm{BDD_{MPS\_r}}$ において、ある要求イベント集合に対応するパスをたどって到達できる終端ノードは、その要求イベント集合が入力された時の遷移先の候補となるマクロプレステートの集合である。そのため、$\mathrm{BDD_{MPS\_r}}$ 上の対応するパスをたどって到達可能な終端ノードが空集合であるような要求イベント集合が存在する場合、システムはその要求イベント集合に対して、仕様を満たしながら応答することができないことを表している。つまり、終端ノードに空集合を含むような $\mathrm{BDD_{MPS\_r}}$

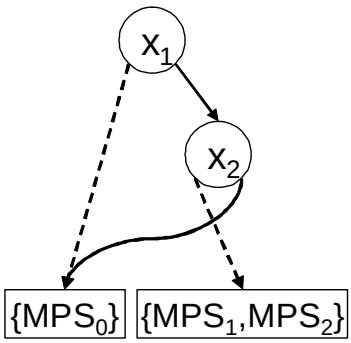

図 4.5: 行き止まりでないマクロプレステートの遷移関係を表す $\mathrm{BDD}_{\mathrm{MPS\_r}}$ の例

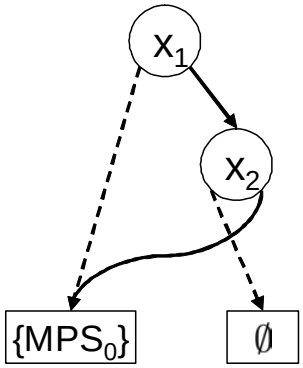

図 4.6: 要求イベント集合 $\{x_1, \neg x_2\}$ に対応する遷移を持たない場合の例

を作るようなマクロプレステートは、行き止まりのマクロプレステートであると判定できる。

たとえば、マクロプレステートから図 4.5 に示すような、終端ノードに空集合を含まない $\mathrm{BDD}_{\mathrm{MPS\_r}}$ が構成された場合、任意の要求イベント集合に対して応答できることを表しており、そのマクロプレステートは行き止まりではない。一方、図 4.6 に示すような終端ノードに空集合を含む $\mathrm{BDD}_{\mathrm{MPS\_r}}$ が構成された場合、要求イベント集合 $\{x_1, \neg x_2\}$ に対する遷移を持たず、そのマクロプレステートは行き止まりである。

したがって、各マクロプレステートに対して $\mathrm{BDD}_{\mathrm{MPS\_r}}$ を計算し、その終端ノードに空集合が含まれているかを確認することで、各マクロプレステートに対する行き止まり判定ができる。

さらに、マクロプレステートグラフ中から行き止まりとなるマクロプレステートを削除し、再び行き止まり判定の処理を行うことを、マクロプレステートグラフが変化しなくなるまで繰り返し行い、初期マクロプ

レステートが行き止まりであるかを判定することで、マクロプレステートグラフの行き止まり判定とする。以下にその手続きを示す。

手続き 4.3 (グラフの行き止まり判定)
マクロプレステートグラフを入力とし、そのマクロプレステートグラフの初期マクロプレステートが行き止まりであれば真を、そうでなければ偽を出力する。本手続きは、以下のステップで処理される。

1. 以下をグラフが変化しなくなるまで繰り返す。
   (a) すべてのマクロプレステートについて $\mathrm{BDD}_{\mathrm{MPS\_r}}$ を計算し、そのマクロプレステートが行き止まりであるか判定。
   (b) 行き止まりと判定されたノードすべてを、マクロプレステートグラフ中から除去。
2. グラフ中に初期マクロプレステートが残っていなければ真を、そうでなければ偽を出力する。

ステップ 1–a では常に $\mathrm{BDD}_{\mathrm{MPS\_r}}$ を再計算する必要がある。それは、ひとつ前のループ開始直後には存在していたマクロプレステートが、そのループ中のステップ 1–b により除去される可能性があるためである。

4.3 二分決定グラフ $\mathbf{BDD_{PS}}$, $\mathbf{BDD_{MPS\_r}}$ を利用した段階的充足可能性判定手続きの実装

手続き 3.4 の実装として手続き 4.1 を、手続き 3.5 の実装として手続き 4.3 を用いることで二分決定グラフ $\mathrm{BDD}_{\mathrm{PS}}$, $\mathrm{BDD}_{\mathrm{MPS\_r}}$ を利用する段階的充足可能性判定手続きを実装できる。

手続き 4.4 ($\mathbf{BDD_{PS}}$, $\mathbf{BDD_{MPS\_r}}$ を利用した段階的充足可能性判定)
動作仕様の段階的充足可能性は以下のステップで判定できる。

1. 動作仕様からプレステートグラフを構成する。
2. ステップ 1 で構成されたプレステートグラフ中に含まれる、すべてのイベンチャリティ違反ノードをグラフから除去する。初期ノードが除去された場合、偽を出力して終了。

3. ステップ 2 で変更されたプレステートグラフをもとに手続き 4.1 を用いて要求、応答イベント集合でグラフを決定化する。

4. マクロプレステートグラフについて手続き 4.3 を用いて要求イベントに対する行き止まり判定をおこない、決定化グラフの初期ノードが行き止まりと判定されれば、偽を出力。そうでなければ真を出力する。

第5章　MPIを用いたタブローグラフ構成手続きの並列化実装

これまで述べてきたように、タブロー法による仕様の性質判定手続きでは、仕様からプレステートグラフなどのタブローグラフを構成し、それらのグラフに対してイベンチャリティ制約などの確認作業を行う。

判定手続き全体に占めるタブローグラフ構成にかかる時間的なコストの割合は非常に高い。青島らにより実装された単一計算機によるタブロー証明器 **T^{3}E** [52] にて付録にあげた簡易 n 階建てエレベータ仕様の例を用いて充足可能性判定を行ったところ、プレステートグラフの構成に 4 階建てエレベータ仕様では 99%もの割合の時間コストがかかっていることが明らかとなった。

また、現実規模の仕様の性質判定を行う場合、構成すべきタブローグラフが巨大になるため、その構成にかかる時間的なコストは非常に大きくなり、1 台の計算機の能力では現実的な時間でグラフ構成処理を終えることができなくなる。

したがって、性質判定手続きの処理時間を短縮し、現実規模の仕様検証を可能とするためには、タブローグラフ構成手続きを高速化する必須があり、そのためには処理の並列化が必要であると考えた。

そこで、本章では 3.1.1 節で紹介したプレステートグラフ構成手続きの処理を並列化して高速化する実装手法について考察を行う。以下では、グラフ構成手続きの並列化アイディアと共に、*Message Passing Interface (MPI)* を用いてそのアイディアを実装する手法について述べる。

5.1 タブローグラフ構成手続きの並列化のアイディア

本節では、プレステートグラフ構成手続きの高速化するにあたり考察したことを述べる。

入力を分割する並列化は効果的か？ プレステートグラフの構成手続きの入力は論理式である。そのため、入力を分割する並列化として考え得るのは、“入力の論理式を部分式に分割し、それぞれの部分式に対応するグラフを構成し、最後に部分式から得られたグラフを合成する” という方法であろう。この方法でのグラフの合成処理は、各グラフのノードの直積をとる形での合成になり、この合成処理はグラフの構成と同程度の手間がかかる。したがって、入力を分割する方法での並列化では効果が期待できないと判断した。

グラフの構成手続きをどのように並列化するか？ ある処理の並列化を行う場合、繰り返し実行されている部分処理に着目し、その部分処理が複数のプロセスで並列に実行されるように実装することが定石となっている。プレステートグラフ構成手続きの場合、式の分解手続きが引数としてとるプレステートを変えながら繰り返し実行されている。そこで、式の分解手続きを処理するプロセスを複数の計算機に用意し、引数の異なる式の分解手続きを並列に実行することでグラフ構成手続きを並列化することとした。また、式の分解手続きの引数や、その処理結果の送受信には、MPI を用いることとした。

構成していくグラフをどのように管理するか？ プレステートグラフ構成手続きにおいて、あるプレステートに対する式の分解手続きは、そのプレステートの次ノード集合を計算することにあたる。式の分解手続きを並列に実行する場合、構成していくプレステートグラフにおいて、どのプレステートについて次ノード集合の計算を行ったか等の情報を管理する必要がある。そこで、本章で紹介する並列化実装では、構成中のプレステートグラフの情報をすべて保持する管理プロセスを用意することとした。

5.2 MPI

メモリ分散型の並列環境では、その環境上で動作する各プロセスは独立したメモリ空間を持つ。メモリ分散型の並列環境上のプロセス間でデータを共有する場合は、そのプロセス間で共有しようとするデータを含んだメッセージの送受信が行われ、データが共有される。メッセージパッシングは、このプロセス間通信の一形態であり MPI はメッセージパッシングの標準 API となっている。MPI は MPI forum [21] によって規格が明確に定義されており、現在の最新版は MPI Version 2.2 として 2009 年にリリースされている。

MPI では、HPF (High Performance Fortran) などと異なり、プログラマ自身が並列処理についての言及をプログラム中に明示的に記述できる。そのため、HPF などに比べプログラマが細かなチューニングが行えるという特徴がある。また、MPI では同一のプログラムを実行する複数のプロセスが、それぞれ異なるデータに対して同時に実行開始され、その実行過程でメッセージ通信を行いながら処理を進めることもできる。これを *Single Program Multi Data (SPMD)* と呼ぶ。

MPI を利用する場合、実行環境上の各プロセスには**ランク**と呼ばれる一意のラベルが設定されるため、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。例えば、ランク 0 のプロセスのみ、他のプロセスと異なる種類のデータを扱うため処理を変える、といったことも単一のプログラムで実現可能である。

MPI はメッセージパッシングの標準規格であり、その実装は様々なベンダーから提供されている。フリーの MPI 実装としては、アルゴンヌ国立研究所で開発されている MPICH2 [27]、インディアナ大学にて管理されている LAM/MPI [42]、LAM/MPI やロスアラモス国立研究所の LA-MPI、テネシー大学の FT-MPI などの開発グループが集まり開発された Open MPI [43] などがよく知られている。商用の MPI 実装には Intel 社の Intel MPI Library [14] や Microsoft 社の Windows HPC Server に同梱されている MS MPI [15] などが提供されている。また、Java 言語用に、mpich2 や LAM/MPI 等のネイティブの MPI 実装とのインターフェイスを提供する mpiJava [36] なども存在する。

5.3 MPI を用いた並列化実装

本節では、5.1 節のアイディアを基にした実装法を示す。本節で提案する実装は、構成するプレステートグラフを管理するプロセス *graphManager* と、次ノードの計算に必要な式の分解手続きを行うプロセス *decomposer* プロセスの 2 種類のプロセスからなる。

グラフ情報を一元管理するため *graphManager* プロセスは実行環境中にただ一つ、式の分解手続きを並列に処理するため *decomposer* プロセスは複数用意する。各 *decomposer* プロセスは、*graphManager* プロセスと MPI を用いて通信し、式の分解手続きに必要な情報とその結果の送受信を行う。

MPI を用いた実装は SPMD の形式をとるため、*graphManager* プロセス、*decomposer* プロセスともに基本的には同じプログラムが実行されることとなる。本実装では、プロセスのランクによりプログラムの処理を分岐させることで 2 種類のプロセスの実行を実現する。

手続き 5.1 (MPI を用いたプレステートグラフ構成手続き)
式 φ を入力とし、プレステートグラフ $\langle V_p, v_{p0}, E_p \rangle$ を出力する。どのプロセスもまず *constPSGraphMPI* メソッドを実行し、それぞれのランクにあわせて *graphManager* メソッド、*decomposer* メソッドのいずれかを実行する。*graphManager* メソッドを実行するプロセス (*graphManager* プロセス) は環境中に 1 プロセスのみである。また、以下の擬似コード中に現れる **send**, **receive** は、MPI 利用した送信、受信処理を表す。

```
1:  constPSGraphMPI(φ : formula)
2:     if (rank is the graphManager's rank)
3:        graphManager(φ);
4:     else
5:        decomposer();

1:  graphManager(φ : formula)
2:     v_p0 := {φ}, V_p := {v_p0}, E_p := ∅, V'_p := ∅;
3:     D_P is the set of all decomposer processes;
4:     T_P is an empty table;
5:     while ( (V_p − V'_p ≠ ∅) ∨ (T_P ≠ ∅) )
6:        while ( (V_p − V'_p ≠ ∅) ∧ (D_P ≠ ∅) )
```

```
7:          p ∈ V_p − V'_p, V'_p := V'_p ∪ {p};
8:          d_P ∈ D_P, D_P := D_P − {d_P};
9:          put (d_P, p) on T_P;
10:         send p to d_P;
11:       receive P from d'_P;
12:       V_p := V'_p ∪ P;
13:       while ( p'' ∈ P )
14:         E_p := E_p ∪ {⟨p', p''⟩ | (d'_P, p') ∈ T_P};
15:       remove (d'_P, p') from T_P;
16:       D_P := D_P ∪ d'_p;
17:     while ( d_P ∈ D_P )
18:       send termination message to d_p;
```

```
1:  decomposer()
2:    m_P is the graphManager process;
3:    while (true)
4:      receive p from m_P;
5:      if (p is the termination message)
6:        break;
7:      P := Temp(decomp(p));
8:      send P to m_P;
```

***constPSGraphMPI* メソッド**　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。

***graphManager* メソッド**　*graphManager* プロセスの行う処理。
3–4 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。T_P は分解作業中のプロセスをキー、そのプロセスで処理されているプレステートを値とする作業割り当て表を表す。
6–10 行目: 現在分解作業を行っていない *decomposer* プロセスに、次ノード計算が未処理のプレステートを割り当てる。
17–18 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。

***decomposer* メソッド**　*decomposer* プロセスの行う処理。
5–6 行目: *graphManager* プロセスから処理終了を知らせるメッセージを受信したら無限ループを抜け処理を終了する。

5.4 実験

本節では、前節までに述べた MPI を用いたプレステートグラフ構成手続きの並列化手法の効果を確認するため、青島らにより与えられた単一計算機によるタブロー証明器 **T³E** [52] を拡張して、MPI を利用するタブローグラフ構成手続きを実装し、拡張前の実装と比較する。比較対象となる T³E は、Java 言語により実装されており、本章で提案した実装手法を用いた拡張は、MPI の API として mpiJava [36] を利用し Java 言語により行った。以下では、拡張前の検証系 T³E と区別するため、拡張後の検証系を **T³EM**（T³E with MPI の略）と記す。本実験では、T³E および T³EM についてプレステートグラフの構成にかかる実行時間を計測し、並列化の効果を確認する。

5.4.1 実験環境

実験には、巻末の付録に添付した簡易 n 階建てエレベータシステム仕様と、n プロセス排他制御システム仕様をそれぞれ複数パターン用意しこれを用いる。これらの仕様から構成されるプレステートグラフの大きさは、表 8.3 に示した通りである。

本実験は、計算機環境として産業技術総合研究所連携検証施設 “さつき”[50] の大規模演算クラスターの計算機ノードを最大 11 台利用して行う。各計算機ノードの 1 台あたりの環境は表 5.2 の通りである。また、以降の章における実験にも、同じ計算機環境を用いるため、表 5.2 には以降の章で利用するソフトウェアについてもあわせて表記している。

5.4.2 実験結果

本実験では、実行環境上にグラフを一元管理する一つの *graphManager* プロセスと、次ノード集合の計算を行う複数の *decomposer* プロセスを用いる。ここでは、一つの計算機ノードあたり一つのプロセスを割り当

表 5.1: 構成されるプレステートグラフの大きさ

	3 階建て	4 階建て	5 階建て
ノード数	88	599	4,247
エッジ数	1,644	22,976	329,476
	8 プロセス	9 プロセス	10 プロセス
ノード数	256	512	1,024
エッジ数	24,057	78,732	255,879

表 5.2: 実験に利用した計算機環境

CPU	Intel Xeon^{TM} E5450 3.0GHz (4 Core) × 2
メモリ	24GB
OS	Debian GNU/Linux 5.0 (64bit)
Java	Version 1.6.0_22 (64bit)
MPICH2	Version 1.4rc (本章の実験で利用)
mpiJava	Version 1.2.5 (本章の実験で利用)
HORB	Version 2.0.2 (6 章、7 章の実験で利用)
Sat4j	Version 2.3.1 (8 章の実験で利用)

てる様にプログラムを実行し、*decomposer* プロセスの数を変化させて処理にかかる時間を計測した。その結果を表 5.3 に示す。この表では台数の欄に *decomposer* プロセスの数を表記した。すなわち台数 3 の行は、*graphManager* プロセスを実行する計算機 1 台、*decomposer* プロセスを実行する計算機 3 台で実行したときの結果を表している。

また、*decomposer* プロセスを複数台利用した T^{3}EM の実行速度が、T^{3}E 単体での実行に比べ何倍の速度になるかを表す速度向上率を、以下の式を用いて求め、それらを表 5.4 にまとめた。また、図 5.1 は、表 5.4 をグラフにおこしたもので、赤の実線は *decomposer* プロセス数と同じだけの速度向上率が得られた場合の取り得る値を表している。

$$\text{速度向上率}\ (N) = \frac{time_{\mathrm{T^3E}}}{time_{\mathrm{T^3EM}}(N)}$$

ただし、$time_{\mathrm{T^3E}}$ は T^{3}E を用いた場合のグラフ構成にかかる時間、$time_{\mathrm{T^3EM}}(N)$ は *decomposer* プロセスを N 台利用したときの T^{3}EM での実行時間とする。

表 5.3: プレステートグラフの構成にかかる時間 (単位:sec)

台数	3 階建て	4 階建て	5 階建て	8 プロセス	9 プロセス	10 プロセス
T^3E	4.64	57.2	2738.7	7.16	45.0	221.8
2	3.32	34.6	1394.8	6.32	28.0	123.8
3	2.48	23.6	929.3	4.63	19.2	87.4
4	2.04	18.3	684.4	3.85	15.0	68.1
5	1.77	14.9	549.9	3.38	12.7	55.0
6	1.61	12.8	475.7	3.04	11.0	47.4
7	1.48	11.2	441.0	2.92	9.6	41.4
8	1.40	10.1	356.2	2.76	8.9	37.7
9	1.34	9.3	318.3	2.59	8.6	34.6
10	1.26	8.6	288.5	2.58	8.0	32.2

表 5.4: T^3E と比較したときの速度向上率

台数	3 階建て	4 階建て	5 階建て	8 プロセス	9 プロセス	10 プロセス
2	1.40	1.65	1.96	1.13	1.60	1.79
3	1.87	2.42	2.95	1.55	2.34	2.54
4	2.27	3.13	4.00	1.86	3.00	3.26
5	2.62	3.84	4.98	2.12	3.55	4.04
6	2.89	4.49	5.76	2.36	4.08	4.68
7	3.13	5.10	6.66	2.45	4.67	5.35
8	3.33	5.66	7.69	2.60	5.05	5.88
9	3.46	6.18	8.60	2.76	5.26	6.41
10	3.68	6.68	9.49	2.77	5.60	6.89

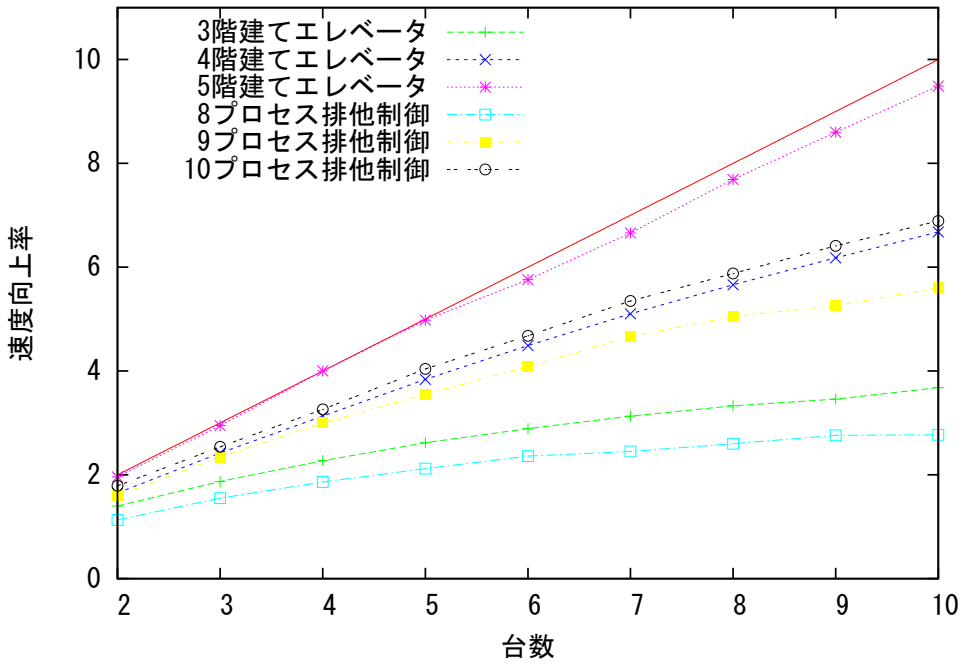

図 5.1: *decomposer* プロセス数と速度向上率

5.4.3 評価

本実験の結果を観察すると、どの入力についても利用台数の増加により実行時間の短縮、速度向上率の上昇がみられ、台数増加の効果が現れていることが分かる。したがって、本章で提案した実装手法が当初の目的通りグラフ構成にかかる時間の短縮を達成できることが示された。

次に、台数効果の大きさについて考察する。T^{3}EM では、*decomposer* プロセスでのみ次プレステート集合の計算が行われ、グラフのエッジを表す各プレステート間のリンクなどのグラフ管理の作業については *graphManager* プロセスの担当となる。ここで単体の T^{3}E において、次プレステート集合の計算にかかる時間の総計を $time_{decomp}$、それ以外にかかる時間を $time_{mngr}$ とする。すると、T^{3}EM で N 台の *decomposer* プロセスを用いたときの次プレステート集合の計算にかかる時間の下限値は $time_{decomp}$ の $1/N$ となる。また、現実的ではないが *graphManager* プロセスが *decomposer* プロセスと完全に並列に動作し、最後の次プレステート集合の計算が終了した時点で *graphManager* プロセスの実行が終了するという状況を考えると、T^{3}EM におけるグラフ構成手続きにかかる時間の下限値

も $time_{decomp}/N$ となる。したがって、そのときの速度向上率の上限値は

$$N\left(1+\frac{time_{mngr}}{time_{decomp}}\right)\ (<N+1)$$

となる。しかし現実的にはこのような理想的な状況はあり得ず、*decomposer* プロセス数分の速度上昇率が得られれば十分に代数効果を発揮していると言って良い。

まず、簡易エレベータ仕様、プロセス排他制御仕様ともに同じ種類の仕様同士で速度向上率を比較すると、どの結果も、構成されるグラフのサイズが大きい仕様の方が速度向上率が良い。また、ほぼ同じ数のノード数を持つタブローグラフを構成する簡易 4 階建てエレベータ仕様と 9 プロセス排他制御仕様の結果を比較するとどの台数の実験でもエッジ数の少ないグラフを構成する簡易 4 階建てエレベータ仕様の方が速度向上率が高い。

これは、まず一つにノード数の多いグラフほど各 *decomposer* プロセスが次ノード計算に要する時間の総計が平均化され、その結果が現れていると考える。また、一つのノードからのエッジ数が少ないほど *decomposer* プロセスが返信する情報が少なくなり、単位時間あたりに *graphManager* プロセスが *decomposer* プロセスから受ける受信回数が増える。そのため、*decomposer* プロセスが送信待ち状態になる時間が減りこの影響が速度向上率に表れたものと考える。結果から、本実装はノード数が多く各ノード間の結合度が小さいタブローグラフを構成するような仕様に適した実装であると結論づける。

第6章　分散オブジェクト技術を用いたタブローグラフ構成手続きの並列分散化実装

5章では、タブローグラフ構成手続きの高速化を狙った並列化を提案した。5章で示した実装では、グラフ情報を管理する *graphManager* プロセスがすべてのグラフ情報を一元管理していた。*graphManager* プロセスがアクセスできるメモリ空間は、そのプロセスが実行されている計算機ノードのメモリ空間に限られる。そのため、どんなに利用する計算機ノードが増加しても *graphManager* プロセスがアクセスできるメモリ空間を超える巨大なグラフはその構成途中でメモリ不足に陥り、正常に終了できないという問題が残っている。そのような巨大なグラフを構成、保持するためには複数の計算機上にグラフ情報を保持させる必要がある。

本章では、プレステートグラフを構成しながらそのグラフ情報を分散させ、実行環境全体でグラフを構成、保持するという方式の実装手法について述べる。本実装手法は、分散オブジェクト技術を用いてグラフ構成の並列処理とグラフ情報の分散配置を同時に実現する。

6.1　並列分散化アイディア

本節では、プレステートグラフを構成中に分散配置し、並列に処理を行うグラフ構成手続きを実現するにあたり考察したことを述べる。

グラフ情報をどの様な単位で分割するか？　プレステートグラフの構成手続きでは、各プレステートについて式の分解手続きを適用し次ノードの計算を行う。そのため、ひとつのプレステートに関する情報を分割す

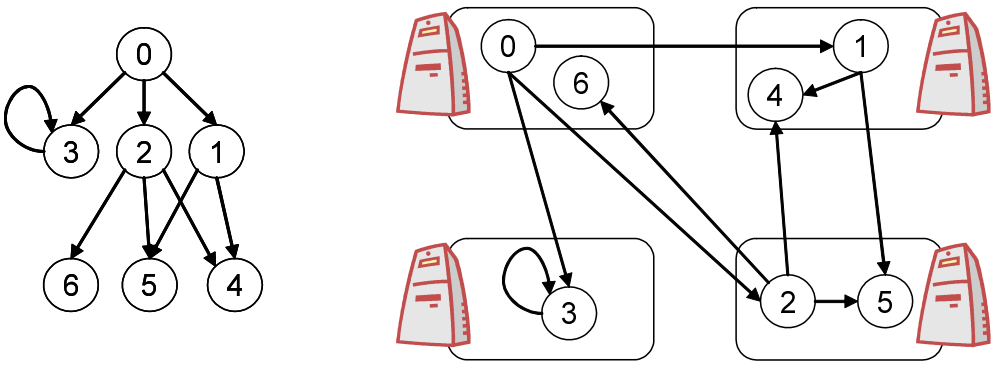

図 6.1: タブローグラフが分散環境上に保持される様子

るような単位でグラフ情報を分割するのは不適切であると考えた。また、MPI を用いた並列化実装においても、プレステート単位での処理を並列化していた。そこで、グラフ情報を複数の計算機ノードに分割してもなお、次ノード集合計算の並列化が容易に行えるように、構成するグラフをプレステート単位で分散配置することとした。

グラフ情報をどの様に分散配置するか？　グラフ情報の分散配置にあたって、同じラベルを持つノードが複数の計算機ノードに保持されないように、ノードのラベルのハッシュ値を用いて、そのラベルを持つノードがどの計算機に配置されるかを決定する。また、グラフのノードオブジェクトは生成されてから特定の計算機に送信されるのではなく、そのノードが配置されるべき計算機上でのみ、そのノードオブジェクトを生成する。これによりノードオブジェクトが重複して生成されていないかなどの管理が容易となる。

ノード間のエッジは、次節で説明する**分散オブジェクト技術**を利用して、異なる計算機上にあるノードオブジェクト間をネットワーク越しにリンクさせることで実現する。これを模式化したものを、図 6.1 に示す。

6.2　分散オブジェクト技術

分散オブジェクト技術は、共通の呼び出し規則に従って動作するソフトウェア部品 (これをオブジェクトと呼ぶ) をネットワーク上の複数の計算機に配置し、それらを連携させて動作させることでシステムを構成する技術である。

システムの機能に応じて部品化することにより、再利用性が高まり、一部を修正しても全体を構成し直す必要が無くなるなどの、開発効率の向上も期待できる。また、分割したソフトウェアを複数の計算機で役割分担をして実行することにより、効率的な資源の活用が可能となる。

代表的な分散オブジェクト技術としては、OGM が定義した Common Object Request Broker Architecture (CORBA) [22] やマイクロソフト社の Distributed Component Object Model (DCOM)、産業技術総合研究所により開発された Java 言語ベースの HORB [40]、Java 言語の標準 API として提供されている Java Remote Method Invocation (Java RMI) などがある。

6.3 分散オブジェクト技術を利用した並列分散化実装

本節では、6.1 節のアイディアを基にした、分散化オブジェクト技術を用いたプレステートグラフ構成手続きの実装手法について述べる。

6.3.1 分散オブジェクト技術の利用

本論文で与える実装法では分散オブジェクト技術を利用する。実装に分散オブジェクト技術を利用することで、リモートに存在するオブジェクトをあたかもローカルに存在するオブジェクトのように扱うことができる。これを実現するため、実行環境上の各計算機には、自身の持つオブジェクトへのアクセスを受け付けるサーバとして働くプロセスを配置する。

リモートのオブジェクトへのアクセスが容易であるため、“グラフ中のノードとその次ノード” のように互いに結びつきの強いオブジェクト同士であっても、同じ計算機上に配置することにこだわる必要が無く、オブジェクトの配置に自由度を持たせることができる。このため、検証手続きの実装において各計算機のもつ分散メモリを自由に扱うことができ、広大な空間を利用することが可能となる。

また、同じ種類のオブジェクトが複数の計算機に分散されているため、それぞれの計算機が個別に自身の持つオブジェクトに対する処理を実行できる。この利点を利用することで、プレステートグラフの構成手続き

や決定化手続きを並列化し、さらに各グラフの構成中にそのノードを分散配置することが容易になる。

6.3.2 プレステートグラフ構成手続きの並列化実装

本章で提案する実装では、実行環境中に処理内容の異なる 2 種類のプロセスを用意する。ひとつは、判定器の処理を統括するプロセスで、主に実行環境中の他のプロセスの処理を監視する目的をもつ。以後このプロセスを**マスタプロセス**と呼ぶ。またマスタプロセスは、構成するプレステートグラフの初期ノードオブジェクトがどの計算機上のプロセスで保持されているかを把握する役割も担う。マスタプロセスは実行環境中にひとつつだけ用意する。

残りのプロセスは、構成中のタブローグラフの一部であるノード集合の保持、管理と、そのプロセスが保持しているノードに対する処理を行う。また、このプロセスは他のプロセスに対して、自身の持つノードオブジェクトに対するアクセスサーバの役割も果たす。以後このプロセスを**ワーカプロセス**、または単に**ワーカ**と呼ぶ。ワーカプロセスは、自身の保持するプレステートに対して次ノード集合を計算する処理も担当する。次ノード集合の計算は各ワーカプロセスで別々に計算されるため、MPI を用いた実装と同様に次ノード集合の計算処理が並列化される。

プレステートグラフの構成処理に関わるすべてのワーカプロセスは構成中のプレステートグラフの一部分を分割してプレステートの集合として保持する。各プレステートオブジェクトは、分散オブジェクト技術を利用して、他のワーカプロセスから接続できるように管理される。また、自身の保持するプレステートで、遷移先のオブジェクトへのリンクが完了していないものについての情報も管理する。

加えて、このプロセスは次の 2 種類のスレッドを生成し実行することでグラフの構成処理を行う。ひとつは自身の保持するプレステートの次ノード集合の計算を行うスレッドで、もうひとつは別のワーカプロセスの持つノードと自身の保持するノード間にリンクを張る処理を行うスレッドである。ひとつのワーカプロセス内では、これらの 2 種類のスレッドが同時に実行される可能性があり、同様に、同じ種類のスレッドも同時に複数実行されうる。

本実装ではこれらのスレッドについての管理を**スレッドプール**の機構を用いて行う。この機構では、あらかじめ設定した数のスレッドオブジェ

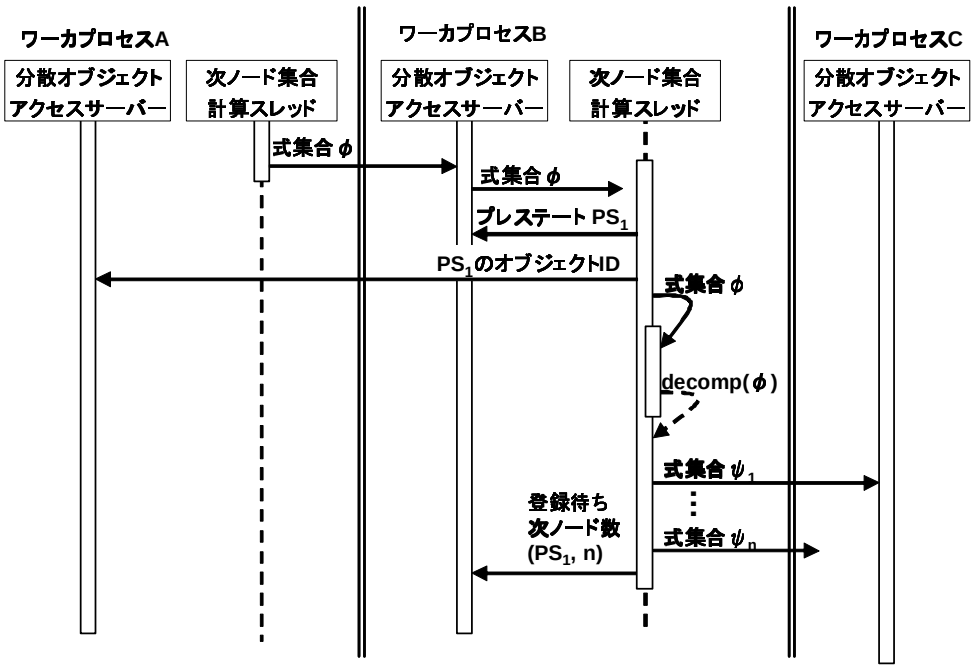

図 6.2: 次ノード集合計算スレッドの処理

クトが用意されプールされる。そしてスレッド処理が必要になった時点で、プールしておいたスレッドオブジェクトを呼び出し、そのオブジェクトにスレッドの処理内容を与え処理を実行させる。処理が終了したスレッドオブジェクトは破棄されることなく、再びプールされ再利用される。

スレッドプールを実装に用いる場合、スレッドに実行させる処理内容を専用のキューに登録しておくだけでよく、スレッド数の管理や処理開始のタイミングについての扱いが容易になる。

以下に、これら 2 種類のスレッドが行う処理を示す。

手続き 6.1 (次ノード集合の計算)
次ノード集合の計算を行うスレッドは以下の手順で処理を実行する。

1. 時間式集合を他のワーカプロセスから受信する。ただし初期ノード生成時のみ、マスタプロセスから動作仕様を表す式集合を受信する。
2. 受信した時間式集合からプレステートを生成し登録する。
3. ステップ 2 で生成したプレステートのオブジェクト ID を時間式集合の送信元に返信する。

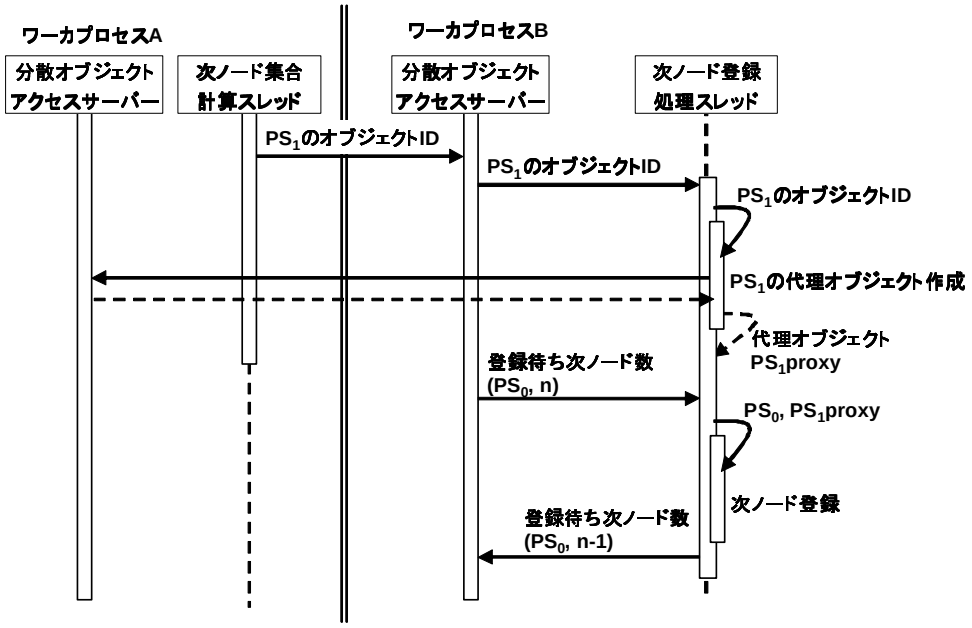

図 6.3: 次ノード登録スレッドの処理

4. 時間式集合に式の分解手続き *decomp* を適用し、得られた集合の各要素を、他のワーカプロセスに送信する。同時に、ステップ 2 で生成したプレステートを、次ノードへのリンク登録待ちの親ノードとして登録し、そのプレステートについての登録待ちの次ノード数を *decomp* の適用結果の要素数に設定する。

図 6.2 は、上記の手続き 6.1 の処理の様子を模式化したものである。図 6.2 において、PS_1 は式集合 ϕ から作られたプレステートオブジェクトを表す。また、式集合 $\psi_1, \ldots, \psi_n$ は $decomp(\phi)$ の要素を表し、n は $decomp(\phi)$ の要素数を表している。

手続き 6.2 (次ノードの登録処理)
他のワーカプロセスからプレステートのオブジェクト ID を受信し、対応する親ノードにそのプレステートを次ノードとして登録するスレッドは、以下のように処理を進める。

1. プレステートのオブジェクト ID を他のワーカプロセスから受信する。
2. そのオブジェクト ID から、対応するプレステートの代理オブジェクトを生成する。

3. 対応するプレステートの親ノードに、代理オブジェクトを次ノードとして登録する。

4. 親ノードが登録待ちをしている次ノードの数を表す値を 1 減らす。

5. その親ノードの登録待ち数が 0 となれば、その親ノードを次ノード登録待ちの集合から登録を抹消する。

図 6.3 は、上記の手続き 6.2 の処理の様子を模式化したものである。図 6.3 において PS_1 はプレステート、PS_0 は PS_1 の親プレステートを表している。また、PS_1proxy はプレステート PS_1 の代理オブジェクトを表している。

これら 2 種類の処理は先に述べたように、スレッドに実行させる処理として専用のキューに追加される。処理内用を追加するキューは、処理ごとに分かれており各処理を担当するスレッドの属するスレッドプールもそれぞれ別に用意する。

6.4 実験

本節では、前節までに述べた分散オブジェクト技術を用いたプレステートグラフ構成手続きの並列分散化手法の効果を確認するため、単一計算機によるタブロー証明器 T^3E を拡張して提案した手続きを実装し、T^3E 並びに 5 章で与えた MPI を用いた実装 T^3EM と比較する。以下では、他の検証系と区別するため、本章で与える分散オブジェクト技術を用いた検証系を **T^3ED**（T^3E with Distributed object technology の略）と記す。T^3ED は T^3EM と同様、Java 言語により実装を行った。その際、分散オブジェクト技術のライブラリとしては、Java 言語と親和性の高い HORB [40] を用いた。

実験環境としては、5.4 節で利用した計算機環境と入力を用いる。実行環境上には一つのマスタプロセスと、複数のワーカプロセスを用意し、一つの計算機ノードあたりに一つのプロセスを割り当てるようにプログラムを実行する。ワーカプロセスの数を変化させ、グラフ構成処理にかかる時間とグラフの構成が終了したときの各ワーカプロセスの消費メモリの平均値を計測する。

6.4.1 実験結果

表 6.1 は、T^{3}ED についてプレステートグラフの構成にかかった時間をまとめたもので、T^{3}E および T^{3}EM での実行時間も合わせて載せている。台数の欄にはワーカプロセスの数を表記している。

また、表 6.3 には、各ワーカプロセスのメモリ消費量を Java 言語標準のプロファイラツール HPROF を用いて計測し、その平均を求めたものをまとめている。ただし、簡易 5 階建てエレベータ仕様と、表中に “—” とあるものについては、HPROF が異常終了するなどして結果が得られなかったことを表している。

6.4.2 評価

表 6.1 および表 6.2 の結果から、残念がら T^{3}ED は T^{3}EM に比べ処理速度の面で劣っていることが分かる。この原因としては、次の 2 つがあげられる。まず一つ目として、T^{3}EM の *decomposer* プロセスが次ノード集合の計算のみを行っているのに対し、T^{3}ED はワーカープロセスは、次ノード集合の計算に加えて、グラフ管理の処理を行っているという点である。ワーカプロセスは、自身の保持するノードへのネットワーク越しのリンク処理を行う。この処理は、自身の保持するノードについて、そのノードの親ノードに対しても、子ノードに対しても行う必要がある。各ワーカープロセスには、この処理時間が加算されることになる。

2 つ目は、T^{3}EM 実装に比べてネットワークを流れるオブジェクトの数が多くなるという点である。T^{3}ED では、プレステートの次ノード集合の計算において、あるラベルを持つプレステートを子ノードに持つことが分かった場合、そのラベルをネットワーク越しに他のプロセスに送信する。この通信は、送信先のプロセスにてすでにこのラベルを持つプレステートが存在するか否かに関係なく行われる。これは T^{3}EM での処理に当てはめると、すでに次ノード計算の処理の終わったプレステートを再度通信路に流すことにあたり、その分の処理時間が加算されてしまう。

また表 6.1、表 6.2 からは、T^{3}ED では必ずしもワーカプロセスの台数の多いものが処理時間が短いとは限らないということが分かる。これは、各ワーカプロセスの保持するプレステートの数に偏りが出てしまうためである。例えば、9 プロセス排他制御システム仕様について観察すると、ワーカプロセスの台数が 5 台と 6 台の時の処理時間がそれぞれ 23.0 秒、30.2 秒となっている。5 台での分散処理実行時には、すべてのワーカプロ

セスが 102 個または、103 個のプレステートを保持し、ほぼ均等に分散されている。一方、6 台での分散処理時には、45、45、85、85、126、126 といった形で偏って分散され、126 個のプレステートを処理するワーカプロセスがボトルネックになってしまい、5 台での実行に比べ多くの時間をかけてしまっていると考える。

ワーカプロセスの保持するプレステートの数が偏ってしまう原因は、プレステートをどのワーカプロセスに割り当てるかを決定するために、ハッシュ関数を用いていることにある。しかし、実行環境上の情報を使うことなく、プレステートを一意のワーカプロセスに保持させるためには、この方法をとる以外はないと考えている。

表 6.3 からは、ワーカプロセスの台数が増えるほど、概ねワーカプロセスの消費するメモリが減る傾向にあることが分かる。これは、メモリ消費の負担を実行環境上のワーカプロセスで分担していることを表し、T^3ED が当初の目的を達成できていることを示している。

しかし、各ワーカプロセスの消費するメモリの総量は T^3E が単体で消費するメモリ数よりも多くなっている。これには、次の 2 つの理由が挙げられる。まず一つ目は、異なるプロセスにあるプレステート同士をつなぐために利用する代理オブジェクトがメモリを消費してしまうためである。n プロセス排他制御システム仕様の様に、一つのプレステートが持つ子ノードの数が多い “密なプレステートグラフ” が構成される仕様では、この影響が強く表れる。

また、2 つめの理由は各ワーカプロセスが個々に BDD を管理していることにある。T^3E およびそれを拡張した T^3EM、T^3ED では、タブローグラフ構成手続きの部分手続き “式の分解手続き” に BDD を利用している。式の分解手続きは次ノード計算処理に必要な手続きで、T^3ED では各ワーカプロセスが別々にこの手続きを処理し、BDD を構成している。そして、各ワーカプロセスは、再度同じ BDD を作成することによる無駄な時間を省くため、一度作成した BDD は再利用のため保持することになる。T^3ED では、各ワーカプロセスの保持する BDD は共有されないため、実行環境上では同じ BDD が複数存在する場合もあり、メモリ消費量の総量を大きくする原因となっている。

しかし、依然として大きなプレステートグラフを構成する必要のある仕様については、利用するプロセス数を増やすことで逐次実行の処理系 T^3E と比べて処理にかかる時間的なコスト、1 台のプロセスが確保する空間的なコストのどちらも抑えることができることは変わらず、本実装は

このような仕様に対して有効であると結論づける。

表 6.1: プレステートグラフの構成にかかる時間 (単位:sec)

	3 階建て		4 階建て		5 階建て	
T^3E	4.64		57.2		2739	
台数	MPI	HORB	MPI	HORB	MPI	HORB
2	3.32	5.10	34.6	58.2	1395	3212
3	2.48	4.10	23.6	40.4	930	1817
4	2.04	3.10	18.3	29.2	684	1319
5	1.77	3.10	14.9	22.8	550	1091
6	1.61	3.11	12.8	20.2	476	938
7	1.48	2.11	11.2	17.2	411	800
8	1.40	2.10	10.1	16.2	356	694
9	1.34	2.10	9.3	15.2	318	609
10	1.26	2.10	8.6	12.8	289	541
	8 プロセス		9 プロセス		10 プロセス	
T^3E	7.16		45.0		221.8	
	MPI	HORB	MPI	HORB	MPI	HORB
2	6.32	11.91	28.0	54.7	123.8	326.2
3	4.63	9.12	19.2	36.8	87.4	247.0
4	3.85	7.52	15.0	29.2	68.1	167.6
5	3.38	6.11	12.7	23.0	55.0	137.6
6	3.04	7.32	11.0	30.2	47.4	119.8
7	2.92	5.12	9.6	18.4	41.4	91.6
8	2.76	4.31	8.9	27.2	37.7	84.4
9	2.59	4.72	8.6	14.2	34.6	90.5
10	2.58	4.32	8.0	14.8	32.2	68.6

表 6.2: T^{3}E と比較した時の速度向上率 (HORB)

	3 階建て		4 階建て		5 階建て	
台数	MPI	HORB	MPI	HORB	MPI	HORB
2	1.48	0.91	1.65	0.98	1.96	0.85
3	1.87	1.13	2.42	1.42	2.95	1.51
4	2.27	1.50	3.13	1.96	4.00	2.08
5	2.62	1.50	3.84	2.51	4.98	2.51
6	2.89	1.50	4.49	2.84	5.76	2.92
7	3.13	2.20	5.10	3.33	6.66	3.42
8	3.33	2.21	5.66	3.54	7.69	3.92
9	3.46	2.21	6.18	3.77	8.60	4.49
10	3.68	2.21	6.68	4.48	9.49	5.06
	8 プロセス		9 プロセス		10 プロセス	
2	1.13	0.60	1.60	0.82	1.79	0.68
3	1.55	0.79	2.34	1.22	2.54	0.90
4	1.86	0.95	3.00	1.54	3.26	1.32
5	2.12	1.17	3.55	1.96	4.04	1.61
6	2.36	0.98	4.08	1.49	4.68	1.85
7	2.45	1.40	4.67	2.45	5.35	2.42
8	2.60	1.66	5.05	1.65	5.88	2.63
9	2.76	1.52	5.26	3.17	6.41	2.45
10	2.77	1.66	5.60	3.04	6.89	3.23

表 6.3: プレステートグラフの構成処理にワーカプロセスが利用するメモリサイズの平均値 (単位:Mbyte)

台数	3 階建て	4 階建て	8 プロセス	9 プロセス	10 プロセス
T^3E	38.7	179.0	25.5	151.1	—
2	19.9	148.0	54.8	113.3	—
3	12.7	189.6	52.5	145.3	530.6
4	16.6	131.2	37.4	88.7	463.6
5	20.2	101.4	36.4	62.3	300.7
6	16.1	81.0	35.3	95.9	309.4
7	18.2	68.1	33.7	53.8	243.8
8	17.6	68.9	36.9	72.7	285.1
9	16.4	59.9	25.5	82.0	227.1
10	14.3	85.3	22.2	63.5	198.0

第7章　二分決定グラフを利用した段階的充足可能性判定手続きの並列分散化実装

5章および6章では、タブローグラフ構成手続きに注目して並列化や分散化について述べた。4章で示した二分決定グラフをを利用した段階的充足可能性判定手続きの実装を並列分散化しようとした場合、その内部手続きであるタブローグラフ構成手続きは6章で示した手法で並列分散化可能である。

さらに本章では、段階的充足可能性判定手続きで用いる、他の内部手続きについても並列分散化可能かどうかを考察し、並列分散化可能な手続きについてはその実装法を示す。

7.1　並列分散化アイディア

これまでに述べたグラフノードを分散化し次ノード集合の計算を並列化する手法は、主に“プレステートグラフ構成手続き”に注目したものであった。段階的充足可能性判定手続きにおけるグラフの決定化手続きは、マクロプレステート毎に次ノードを計算するという点でプレステートグラフ構成手続きとよく似ている。そのため、グラフの決定化手続きについてもプレステートグラフの構成手続きと同様の手法を用いて次ノード計算の並列処理と、マクロプレステートグラフの分散配置が行える。

また、マクロプレステートグラフがノード単位で分散配置されていれば、行き止まり判定手続きも並列化できる。行き止まり判定手続きは、マクロプレステート単位で処理を行うため、各計算機が自身の持つマクロプレステートに対する処理を別々に行うことで並列化できる。

イベンチャリティ違反ノード除去手続きは、構成されたプレステートグラフを初期プレステートから順にたどって処理を行う手続きであるた

め、これまでに述べた方法では並列化できない。この手続きに関しては並列化せず、ノードが分散化されたグラフをたどって処理が行えるように実装する。

7.2 グラフ決定化手続きの並列分散化実装

本節では、手続き 3.4 で示した $\mathrm{BDD_{PS}}$ を利用する手続きを基にしたグラフ決定化手続きの並列分散化実装について述べる。ここでは、プレステートグラフは 6.3 節で与えたプレステートグラフ構成手続きの並列分散化実装を用いて構成され、すでに実行環境上に分散配置されているものとする。

決定化手続きの並列分散化実装では、プレステートグラフ構成手続きに用いたワーカプロセスを再利用し、各ワーカプロセスは次の 4 つの処理を担当する。

- 自身が保持するマクロプレステートの管理。
- 次ノードの登録を待つ親マクロプレステートの管理。
- 次マクロプレステート集合を計算するスレッドの生成、管理。
- マクロプレステートに次ノードを登録するスレッドの生成、管理。

また、本実装では通信の負荷を下げるため、プレステートの送受信に変えて、そのプレステートの ID を送受信することで、同等の処理を行っている。したがって、手続き 3.4 で利用する $\mathrm{BDD_{PS}}$ を次で定義する BDD_{PS_ID} を用いた操作に変更する。

定義 7.1 ($\mathbf{BDD_{PS\_ID}}$)
BDD_{PS_ID} とは、BDD と同様に二分決定グラフである。非終端ノードのラベルは要求変数および応答変数のみであり、終端ノードはプレステートのオブジェクト ID の集合である。つまり、対応する $\mathrm{BDD_{PS}}$ の終端ノードに含まれるすべてのプレステートをそのオブジェクト ID で置き換えたものである。

グラフ決定化手続きについても、次マクロプレステート集合の計算処理と、別のワーカプロセスが保持するマクロプレステートとの間のリン

ク処理の二種類の処理をスレッドプール機構を用いて実装する。次に、これら二種類のスレッドが行う処理について以下に示す。

手続き 7.1 (次ノード集合の計算)
次ノード集合の計算を行うスレッドは以下の手順で処理を行う。BDD_{PS_ID} のひとつの終端ノードはプレステートの ID 集合であり、その ID 集合から作られるマクロプレステートが、現在のスレッドで生成したマクロプレステートの次ノードとなる。

1. プレステートの ID 集合を他のワーカプロセスから受信する。
2. プレステートの ID 集合からマクロプレステートを生成し登録する。同時に、次ノードの登録待ちの親ノードとしても登録する。
3. ステップ 2 で生成したマクロプレステートの ID をプレステートの ID 集合の送信元に返信する。
4. プレステート ID の集合から、そのマクロプレステートの次ノード集合を表す BDD_{PS_ID} を生成する。
5. BDD_{PS_ID} に現れる各終端ノードに対応するプレステートの ID 集合を他のワーカプロセスに送信する。

手続き 7.2 (次ノードの登録処理)
マクロプレステートを次ノードとして自身の親ノードに登録するスレッドは、以下の手順で処理を行う。

1. マクロプレステートの ID を他のワーカプロセスから受信する。
2. 受信した ID から、そのマクロプレステートの代理オブジェクトを生成する。
3. 代理オブジェクトを、対応するマクロプレステートの親ノードに、次ノードとして登録する。
4. 親ノードが登録待ちをしている次ノードの数を表す値を 1 減らす。
5. 登録待ちの次ノード数が 0 となれば、その親ノードを次ノード登録待ちの集合から登録を抹消する。

7.3　行き止まり判定手続きの並列分散化実装

次に、前節で与えた決定化手続きによって、分散環境上に構成されたマクロプレステートに対する行き止まり判定手続きの実装手法を与える。

本節で与える実装手法では、以下で与える手続き 7.4、手続き 7.5 の処理を、分散環境上のワーカプロセスそれぞれが並列に実行する。これらの処理では、各ワーカは自身が保持するマクロプレステートに対する処理を担当する。したがって、別のワーカの保持するマクロプレステートとは独立に行き止まり判定、およびノード除去が並列に処理される。また、これらの処理を統括するためマスタプロセスは、以下の手続き 7.3 を実行する。

手続き 7.3 (分散化グラフの行き止まり判定)
分散化されたマクロプレステートグラフの行き止まり判定は次の手順で処理する。

1. 以下を初期マクロプレステートが行き止まりとなるまで繰り返す。
   (a) 各ワーカプロセスに手続き 7.4 の行き止まり判定処理を開始させる。
   (b) 各ワーカの判定結果の論理和をとる。
   (c) その論理和が偽ならばループを抜ける。
   (d) 各ワーカに手続き 7.5 の行き止まり除去処理を開始させる。
   (e) すべてのワーカの行き止まり除去処理の終了を待つ。
2. 初期マクロプレステートが行き止まりならば真を出力し、そうでなければ偽を出力する。

手続き 7.4 (各ワーカでの行き止まり判定処理)
各ワーカプロセスは、自身が保持するマクロプレステートに対して、次の手順で行き止まり判定処理を行う。

1. 自身の計算機で保持するすべてのマクロプレステートに対して以下を行う。
   (a) マクロプレステートから $\mathrm{BDD}_{\mathrm{MPS}\_\mathrm{r}}$ を計算する。

(b) その BDD_{MPS_r} の終端ノードのラベルに空集合が含まれているか判定する。

(c) 空集合が含まれていた場合、そのマクロプレステートを行き止まりとして登録する。

2. ステップ 1 で、行き止まりのマクロプレステートが発見された場合、マスタプロセスに真を送信し、発見されなかった場合は偽を送信する。

手続き 7.5 (各ワーカでの行き止まり除去処理)
上述の手続き 7.4 で行き止まりと判定されたマクロプレステートを除去する。

1. 手続き 7.4 において行き止まりと判定されたマクロプレステートをすべて除去する。

2. 行き止まりのマクロプレステートをすべて除去したことをマスタプロセスに報告する。

7.4 実験

前節までの並列分散環境上への実装手法を適用し、Java 言語および、分散オブジェクトのライブラリ HORB を用いて T^3ED に段階的充足可能性判定手続きを実装した。本節では、実験により段階的充足可能性判定手続きの並列化の効果を確認する。

実験環境としては、5.4 節で利用した計算機環境と入力を用いる。また 6.4 節同様、実行環境上には一つのマスタプロセスと、複数のワーカプロセスを用意し、一つの計算機ノードあたりに一つのプロセスを割り当てるようにプログラムを実行する。ワーカプロセスの数を変化させ、各動作仕様についての段階的充足可能性判定にかかる時間計測する。実験に使用した仕様においてグラフ決定化手続きを経て構成されるマクロプレステートグラフの大きさは、表 7.1 に示した通りである。また、これらの動作仕様はすべて段階的充足可能な仕様である。

表 7.1: 構成されるマクロプレステートグラフの大きさ

	3 階建て	4 階建て	5 階建て
ノード数	88	487	1,804
エッジ数	1644	19,637	99,026
	8 プロセス	9 プロセス	10 プロセス
ノード数	256	512	1025
エッジ数	24,057	78,732	255,882

7.4.1 実験結果

表 7.2 は、T^{3}ED について、段階的充足可能性判定にかかった時間をまとめたものである。ただし、メモリ不足で判定時間を計測できなかった場合には、“O/M”[1] と表記している。また、台数の欄にはワーカプロセスの数を表記している。

本実験では、他の節の実験とは異なり “ワーカプロセスを n 台利用した時の判定速度が 1 台利用時の何倍であるか” を速度向上率と呼ぶこととする。これは、先に述べたとおり T^{3}E に実装されている段階的充足可能性判定手続きが、完全な手続きではないためである。本実験の速度向上率は以下の式で与えられる。

$$\text{速度向上率}\,(N) = \frac{time(1)}{time(N)}$$

ただし、$time(1)$ はワーカプロセス 1 台、$time(N)$ はワーカプロセス N 台用いた時に、段階的充足可能性判定手続きにかかる実行時間とする。その結果を表 7.3 に示す。ただし、メモリ不足により利用する計算機が 1 台の時の判定時間が計測できなかった場合、正しく判定できる最小のワーカプロセス数での結果を基準とする値を括弧内に記した。このときの値は、基準として選んだ箇所の値が、そこで利用した計算機の数 M と等しくなるよう次の計算法を用いて算出した値である。

$$\text{速度向上率}\,(N) = \frac{time(m)}{time(N)} \times M$$

[1] Out of Memory の意

表 7.2: 段階的充足可能性判定にかかる時間 (単位:sec)

台数	3 階建て	4 階建て	5 階建て	8 プロセス	9 プロセス	10 プロセス
1	79.3	1,124	O/M	1,093	O/M	O/M
2	38.3	522	2,418	538	O/M	O/M
3	24.2	366	1,464	342	1,400	O/M
4	19.0	271	1,115	251	1,054	O/M
5	16.6	222	907	202	843	O/M
6	14.6	194	767	165	690	O/M
7	14.8	177	662	154	642	O/M
8	12.4	150	589	129	493	O/M
9	11.7	140	533	121	491	2,093
10	11.3	129	486	108	434	1,948

7.4.2　評価

表 7.2 から、どの仕様に対する実験結果についても、利用する計算機の台数が増えるほど、段階的充足可能性判定にかかる時間が抑えられていることがわかる。また、5 階建てエレベータ仕様や、9 プロセス排他制御、10 プロセス排他制御の仕様などメモリ不足により少ない台数では判定できなかった仕様についても、ある程度台数を増やすことで判定可能となったことが分かる。

この結果から、本実装が当初の目的通り、利用する計算機の数を増やすことで判定にかかる時間を抑えることと、より大きなサイズのタブローグラフを構成することが可能であることが示された。

次に表 7.3 から、どの実験結果においても、利用する台数が増えるにしたがって速度向上率が増加しているのが分かる。そのため、ある程度の台数の計算機が用意できるのであれば、本論文で提案した並列分散化実装が速度向上に有用であるといえる。

3 階建てエレベータ仕様や 4 階建てエレベータ仕様では、速度向上率が、期待する台数効果に比べて小さいものとして現れているものがある。これには、いくつかの要因が考えられるが、主な要因としては次ノード計算のステップに BDD を用いているという点が挙げられる。

プレステートグラフを構成する場合、利用する計算機の数がいくつであっても、初期ノードから次ノードを計算するのにかかる時間はほとんど変わらない。計算機 1 台での実行では、この時に計算に用いられた BDD

表 7.3: ワーカプロセス 1 台に対する速度向上率

台数	3 階建て	4 階建て	5 階建て
2	2.07	2.15	(2 (基準値))
3	3.27	3.07	(3.30)
4	4.17	4.16	(4.34)
5	4.77	5.06	(5.33)
6	5.42	5.78	(6.31)
7	5.36	6.36	(7.30)
8	6.40	7.51	(8.21)
9	6.79	8.04	(9.07)
10	6.99	8.74	(9.95)
	8 プロセス	9 プロセス	10 プロセス
2	2.03	–	–
3	3.19	(3 (基準値))	–
4	4.36	(3.99)	–
5	5.42	(4.99)	–
6	6.64	(6.09)	–
7	7.08	(6.55)	–
8	8.46	(8.52)	–
9	9.06	(8.55)	(9 (基準値))
10	10.16	(9.69)	(9.67)

は以降のどのプレステートの次ノード計算ステップにも再利用可能である。しかし、複数の計算機ノードを用いた実行では、次ノード計算ステップで用いる BDD について、実行環境上の計算機間で共有していないため、他の計算機上で利用された BDD と同様の BDD をそれぞれの計算機が個別に構成してしまう可能性がある。この点が不利に働くと考える。

本論文で提案した次ノードの分配方法は、*decomp* の結果が計算し終わってから、その要素を他の計算機に分配するというものであった。しかし、この実装では初期ノードの *decomp* を計算し終わるまで並列処理が行われず、この処理時間が長くなる大きな仕様では並列化の利点が大きく損なわれることとなる。この点も、速度向上率の向上を妨げていると考える。

しかし、またその逆に 8 プロセス排他制御仕様など速度向上率が、期

待する台数効果よりも大きい値として現れているものも存在する。これは、利用する計算機の台数が少なくなるほど保持するグラフ情報が増え、ノードの重複管理の負荷が大きくなることや、グラフ情報がメモリを圧迫するために頻繁にガベージコレクションが引き起こされるために起こると考える。

以上の実験結果および考察から、本実装手法は、保持するタブローグラフを分散させることで1台のプロセスがグラフ情報の保持に費やすメモリ空間を抑えることができるため、入力される仕様によっては期待される台数効果以上の速度向上率を上げたり、少ないプロセス数ではメモリ不足で判定手続きが正常に終了しなかった仕様についても、判定を可能にするなど、段階的充足可能性判定手続きの実装法として有効な手法であると結論づける。

第8章　SAT solverを用いた式の分解手続きの実装

これまで述べてきたように、プレステートグラフの構成手続きは、その内部で式の分解手続きを用いて次プレステートの計算を行っている。ここまでに提案してきたグラフ構成手続きの実装で用いていた式の分解手続きは、青島らにより提案された実装手法 [52] を用いて実装されたものであり、そこではBDDを利用していた。この実装では式の分解手続きをBDD演算で表現することにより、過去に行ったBDD演算の結果を記憶し、その情報を再利用することで式の分解手続きの実行効率をあげている。

このBDDを利用した実装は、プレステートに対しBDDを利用した式の分解手続きを適用することで、次ノードの候補すべてが一度に得られるという特徴を持つ。しかし、この特徴のためBDDの生成に時間がかかる場合、いくら並列化を行ったところで、その次ノードに対する処理はすべての次ノードが判明した後でなければ開始できない。特に、グラフの初期ノードに対するBDDの生成に時間がかかる場合、5章および6章で示したどちらの実装法を用いたとしても、その間は全く並列に処理を行っていないことになる。複雑な仕様に対する性質判定処理を、並列処理により高速化しようとする場合、この点は非常に不利に働く。

そこで、すべての次ノードの生成の完了を待たずに、利用可能な次ノードを高速に順次生成するための式の分解手続きの実装手法が望まれる。本章では、そのような式の分解手続きの実装法として、高速に命題論理の充足可能性問題（**SAT問題**）を解くツール *SAT solver* を利用した実装法を提案する。

高速なSAT solverの恩恵を得ようと、様々な問題をSAT問題に帰着しSAT solverを利用してそれらの問題の解を高速に得ようとする研究が数多くなされている [13][4][38]。本論文でも、SAT solverの高速性をグラフの構成手続きの高速化に利用する。

本章で提案する実装手法は、タブローグラフ構成において、あるノードからその次ノード集合を計算する問題にSAT solverを利用するものである。ここでは、構成されるグラフのサイズを抑えるため、生成されるノードがドントケアとなる命題変数を多く含むノードとなるよう、SAT solverの入力形式への変換を工夫したり、その入力の下でSAT問題の**極小モデル生成手続き** [46] を利用することを提案する。

そして、本実装手法を用いた性質判定器と、BDDを利用した実装との比較実験を行い、本実装手法の有用性を示す。

8.1 SAT solver

SAT問題は、与えられた命題論理式を真とするような命題変数の真偽値割り当てが存在するか（充足可能であるか）を判定する問題である。また、SAT問題は代表的なNP完全問題としてよく知られている。SAT solverはSAT問題が充足可能であるかを高速に判定するツールで、充足可能と判定した場合にはその証拠となる真偽値割り当てを出力する。

SAT solverについては、その高速化に関する研究が数多くなされている。DPLLアルゴリズム [17] をはじめとして、矛盾解析による学習節を利用するConflict Driven Clause Learning（CDCL）アルゴリズム [41] や二つのリテラル監視によるブール制約伝播（Boolean Constraint Propagation, BCP）の最適化手法 [31] などが提案され、実装されている。現在、高速なSAT solver実装としては、MiniSAT [18], PicoSAT [7] 等がよく知られている。またJava言語による実装としてSat4j [6] が提供されている。

SAT solverは**乗法標準形**（*Conjunctive Normal Form, CNF*）の形式で表現された命題論理式を入力としてとる。入力の論理式が充足可能であればその論理式を充足させる命題変数の真偽値割り当てを一つ出力する。SAT solverにより出力される真偽値割り当ては**モデル**と呼ばれる。モデルは、入力の論理式中に現れるすべての命題変数についてのドントケアを含まない真偽値割り当てとなっている。

SAT solverを利用する場合、論理式中に現れる命題変数を1以上の自然数でラベル付けし、これを用いて論理式をエンコードしたフォーマットを入出力に用いる。入力データはCNF形式の論理式をエンコードした形式で、CNFの節ごとに改行を用いて区切るフォーマットを用いる。CNFの各節は整数の列で表現され、次のようにエンコードされる。

- 各命題の肯定的な出現は、その命題のラベルの値をそのまま用いて表現する。
- 各命題の否定的な出現は、その命題のラベルにマイナス記号を付して表現する。
- 節の終端を 0 を用いて表す。

たとえば、命題変数 x, y, z にそれぞれ 1, 2, 3 を割り当てた時、CNF 形式の命題論理式 $(x \vee \neg y) \wedge (\neg x \vee y \vee z)$ を入力のフォーマットで書き表すと図 8.1 のようになる。

```
c comment
c num of prop = 3, num of clause = 2
p cnf 3 2
1 -2 0
-1 2 3 0
```

図 8.1: SAT solver の入力フォーマットの例

入力として与えられた SAT 問題が充足可能である場合、SAT solver は次のように整数の列としてエンコードされた形式でモデルを出力する。

- モデル中で真に割り当てられた命題変数は、ラベルの値をそのまま用いて表現する。
- モデル中で偽に割り当てられた命題変数は、ラベルの値にマイナス記号を付して表現する。
- モデルを表す整数列の終端を 0 で表す。

図 8.1 の入力を SAT solver に与えると図 8.2 のような出力が得られる。図 8.2 は、入力された命題論理式が充足可能で、そのモデルの一つが x, y, z すべての命題変数に偽を割り当てるものであることを表している。また、このモデルは命題論理式 $\neg x \wedge \neg y \wedge \neg z$ を表しているとみなすこともできる。そのため、SAT solver の出力は、節を一つしか持たない**加法標準形**（*Disjunctive Normal Form, DNF*）の形式で表現された命題論理式であるとみなせる。

```
s SATISFIABLE
v -1 -2 -3 0
c Total wall clock time (in seconds) : 0.023
```

図 8.2: SAT solver の出力の例 (抜粋)

8.2 実装の基本アイディア

先に 3.1 節で述べたとおり、ある式集合に式の分解手続きを適用すると、その結果は論理式集合の集合として表される。ひとつの式集合は、式の分解手続きの意味するところから、その式集合中に現れる論理式すべてを論理積でつないだ論理式と見なすことができる。さらに、それら式集合の集合は、上記の論理式集合を表す論理積でつないだ論理式を、論理和で結んだ式と見なすことができる。

たとえば、論理式 $\Box(r \to \Diamond s)$ に式の分解手続きを適用すると、

$$\{\{\neg r, \Box(r \to \Diamond s)\}, \{s, \Box(r \to \Diamond s)\}, \{\neg s, \Diamond s, \Box(r \to \Diamond s)\}\} \qquad (8.1)$$

となるが、この結果は、次の論理式

$$(\neg r \wedge \Box(r \to \Diamond s)) \vee (s \wedge \Box(r \to \Diamond s)) \vee (\neg s \wedge \Box(r \to \Diamond s) \wedge \Diamond s) \quad (8.2)$$

と見なすことができる。ここで、ある時間式が肯定的に現れることを表す新たな命題変数を用意し、これを時間式の代わりに用いて、この時間式を含んだ論理式を命題論理式の形に変形する。新しく用意する命題変数を**時間式変数**と呼ぶ。各論理式中に現れる否定演算子が先頭に現れない $[f]g$ の形の式を時間式変数 $x_{[f]g}$ で、否定演算子が先頭に現れる $\neg[f]g$ の形の式を $\neg x_{[f]g}$ で置き換える。たとえば論理式 (8.2) は時間式変数を用いて

$$(\neg r \wedge x_{\Box(r \to \Diamond s)}) \vee (s \wedge x_{\Box(r \to \Diamond s)}) \vee (\neg s \wedge x_{\Box(r \to \Diamond s)} \wedge \neg x_{\Box\neg s}) \qquad (8.3)$$

と変形される[1]。

式 (8.1) の様な式集合の集合を引数とし、式 (8.3) の様な時間式変数を含む命題論理式への変換を行う関数を $conv$ とし、以下のように定義する。

[1] $\Diamond s = \neg\Box\neg s$

定義 8.1 (式集合集合から命題論理式への変換 $conv$)
要素がリテラルまたは時間式のみからなる式集合を要素とする集合を引数とし、命題論理式を返す関数 $conv$ を次のように定義する。ただし、以下では p を命題変数、f, g は式とする。$conv_F$ は、リテラルまたは時間式を引数とし命題論理式を返す関数、$conv_S$ は、要素がリテラルまたは時間式である式集合を引数とし命題論理式を返す関数で、ここで同時に定義する。また、S は要素がリテラルまたは時間式のみの式集合、Σ はそのような式集合の集合とする。

$$conv_F(p) = p,\ conv_F(\neg p) = \neg p$$
$$conv_F([g]f) = x_{[g]f},\ conv_F(\neg[g]f) = \neg x_{[g]f}$$

(ただし $x_{[g]f}$ は異なる式ごとに新しい命題変数（時間式変数）とする。)

$$conv_S(S) = \bigwedge_{f \in S} conv_F(f) \tag{8.4}$$
$$conv(\Sigma) = \bigvee_{S \in \Sigma} conv_S(S) \tag{8.5}$$

この変換関数 $conv$ に関して次の定理が成り立つ。

定理 8.1 (関数 $conv$ の性質)
要素がリテラルまたは時間式のみの式集合を要素とする式集合の集合 $\Sigma_i\ (1 \leq i \leq n)$ について

$$conv\left(\bigsqcup_{1 \leq i \leq n} \Sigma_i\right) = \bigwedge_{1 \leq i \leq n} conv(\Sigma_i)$$

が成り立つ。ただし

$$\bigsqcup_{1 \leq i \leq n} \Sigma_i = \left\{ \bigcup_{1 \leq j \leq n} S_j \;\middle|\; 1 \leq j \leq n \ \wedge\ S_j \in \Sigma_j \right\} \tag{8.6}$$

とする。

証明

関数 $conv, conv_S$ の定義から、

$$
\begin{aligned}
& conv\left(\bigsqcup_{1\leq i\leq n}\Sigma_i\right) \\
=& conv\left(\left\{\bigcup_{1\leq j\leq n}S_j \;\middle|\; 1\leq j\leq n \ \wedge\ S_j\in\Sigma_j\right\}\right) \\
& \qquad (\because \text{式 (8.6) より}) \\
=& \bigvee_{S_1\in\Sigma_1,\ldots,S_n\in\Sigma_n} conv_S\left(\bigcup_{1\leq j\leq n}S_j\right) \\
& \qquad (\because \text{式 (8.5) より}) \\
=& \bigvee_{S_1\in\Sigma_1,\ldots,S_n\in\Sigma_n}\left(\bigwedge_{f\in\bigcup_{1\leq j\leq n}S_j} conv_F(f)\right) \\
& \qquad (\because \text{式 (8.4) より}) \\
=& \bigvee_{S_1\in\Sigma_1,\ldots,S_n\in\Sigma_n}\left(\bigwedge_{1\leq j\leq n}\left(\bigwedge_{f\in S_j} conv_F(f)\right)\right) \\
& \qquad (\because \wedge\text{の結合則より}) \\
=& \bigvee_{S_1\in\Sigma_1,\ldots,S_n\in\Sigma_n}\left(\bigwedge_{1\leq j\leq n} conv_S(S_j)\right) \\
& \qquad (\because \text{式 (8.4) より}) \\
=& \bigwedge_{1\leq k\leq n}\left(\bigvee_{S\in\Sigma_k} conv_S(S)\right) \\
& \qquad (\because \wedge,\vee\text{の分配則より}) \\
=& \bigwedge_{1\leq k\leq n} conv(\Sigma_k) \qquad (\because \text{式 (8.5) より})
\end{aligned}
$$

となり、

$$conv\left(\bigsqcup_{1\leq i\leq n}\Sigma_i\right)=\bigwedge_{1\leq i\leq n} conv(\Sigma_i)$$

が成立する。 ■

また、定理 8.1 から、プレステート P の式分解結果と関数 $conv$ に関して次の定理が成立する。

定理 8.2 (プレステート P の式分解結果と関数 $conv$)
あるプレステート P 全体に式の分解手続きを適用した結果を Σ_P、$t \in P$ なる論理式 t のみを含む式集合 $\{t\}$ を式分解した結果を $\Sigma_{\{t\}}$ とすると、

$$\bigwedge_{t \in P} conv(\Sigma_{\{t\}}) = \bigvee_{S \in \Sigma_P} conv_S(S) \tag{8.7}$$

が成り立つ。

証明

式の分解手続き、および定理 8.1 より次の式が成り立つ。

$$\begin{aligned} conv(\Sigma_P) &= conv\Big(\bigsqcup_{t \in P} \Sigma_{\{t\}}\Big) && (\because \text{式の分解手続きより}) \\ &= \bigwedge_{t \in P} conv(\Sigma_{\{t\}}) && (\because \text{定理 8.1 より}) \end{aligned}$$

また、$S \in \Sigma_P$ なる式集合 S を用いると

$$conv(\Sigma_P) = \bigvee_{S \in \Sigma_P} conv_S(S)$$

が成り立ち、この二つの式から

$$\bigwedge_{t \in P} conv(\Sigma_{\{t\}}) = \bigvee_{S \in \Sigma_P} conv_S(S)$$

が導かれる。 ■

ここで、$S \in \Sigma_P$ であることから $conv_S(S)$ は、リテラルが論理積でのみで結ばれた命題論理式となることに注意すると、式 (8.7) は、“各命題論理式 $conv_S(S)$ は $\bigwedge_{t \in P} conv(\Sigma_{\{t\}})$ を満足する SAT 問題のモデルの一つを表す論理式である” ことを示している。

したがって、プレステート P を式分解して $S \in \Sigma_P$ なる式集合 S をすべて獲得することは、プレステートの各論理式 t について式の分解手続きを適用した結果を変換 $conv$ にて命題論理式に変換し、それらを論理積で結んだ命題論理式 $\bigwedge_{t \in P} conv(\Sigma_{\{t\}})$ を充足する SAT 問題のモデルを不足なく発見することに相当する。

本章では、このモデルの発見に SAT solver を用いることで、式の分解手続きを実現する実装法について提案する。

8.3 SAT solver の入力形式への変換

8.2 節で提案した、論理式集合の集合を命題論理式に変換する関数 *conv* を用いて、プレステートに含まれる各論理式に式の分解手続きを適用した結果を命題論理式で表現する。そして、これらの式をすべて論理積で結び、SAT solver の入力とする。論理式 (8.3) では、式の分解手続きの結果 (8.1) との対応を明確にするため、分解結果の論理式を DNF 形式で示した。しかし、SAT solver は CNF 形式の論理式を入力とするため、SAT solver を用いる場合にはプレステート中の各論理式に対する分解結果は、CNF 形式で表現されなければならない。例えば、式 (8.3) は、CNF 形式に変換され以下の式 (8.8) の様に表現される。

$$(\neg r \vee s \vee \neg s) \wedge (\neg r \vee s \vee \neg x_{\Box\neg s}) \wedge x_{\Box(r \rightarrow \Diamond s)} \tag{8.8}$$

空間的な効率の観点から、構成されるタブローグラフはサイズが小さく抑えられていることが望ましい。構成されるグラフの大きさは、式の分解手続きの結果として得られるプレステートの数に影響を受ける。したがって、式の分解手続きの実装法を工夫し、出力されるプレステートの数を少なくすることで空間的な効率化が図れる。

本章の以降の節では、式の分解手続きに SAT solver を利用する場合に用いる式の分解手続きの実装法について述べる。本章で提案する実装法は、分解手続きの結果として得られるプレステートの数を少なくする工夫が施されている。本節の以降の部分では、SAT solver への入力に対して行う工夫について述べる。

直接、式 (8.8) の様な論理式をそのまま SAT solver の入力として用いると、出力されるモデルの数が非常に多く、それに伴い式の分解手続きで得られるプレステートの数が非常に多くなってしまう。これは、SAT solver により出力されるモデルが “論理式中に現れるすべての命題変数についての値割り当て” を表しており、真偽値の割り当てがドントケアになり得る命題変数についても真偽値が必ず割り当てられてしまうためである。

そこで本論文では、ドントケアとなる命題変数の取り扱いを容易にするための工夫として、元の命題変数一つにつき二つの新たな命題変数を用意し、元の式をこれらの命題変数のみからなる式へと変換し、SAT solver の入力とする方法を提案する。

本章で提案する論理式変換法では、変換前の論理式に現れる命題変数 p について、二つの新たな命題変数 p_{po}, p_{no} を用意する。p_{po} は変換元の

表 8.1: 命題 p_{po}, p_{no} の真偽値と、その命題 p に関する意味

p_{no} \ p_{po}	真	偽
真	矛盾	p は偽
偽	p は真	p はドントケア

CNF 式の節に p が肯定的に現れることを表す命題であり、p_{no} は変換元の CNF 式の節に p が否定的に現れることを表す命題である。

これら二つの命題を用いると、ドントケアを含む命題変数 p に関する意味を表 8.1 のように p_{po}, p_{no} を用いて表すことができる。論理式の変換は、変換元の肯定リテラル p を p_{po} に、否定リテラル $\neg p$ を p_{no} に置き換えることで行われる。

しかし、これだけでは SAT solver の出力する解として p_{po}, p_{no} がともに真となる解が出力される恐れがある。p_{po}, p_{no} がともに真となる解は、表 8.1 に示した通り矛盾を表し利用に適さない。そこで、本変換方法ではさらに p_{po}, p_{no} が同時に真とはならないことを表す式 $(\neg p_{po} \vee \neg p_{no})$ を、変数置き換え後の式に CNF 節として追加する。

この式変換法を用いると、式 (8.8) は、次の式 (8.9) の様に変換される。

$$
\begin{aligned}
&(r_{no} \vee s_{po} \vee s_{no}) \wedge (r_{no} \vee s_{po} \vee x_{\Box\neg s\ no}) \\
&\wedge x_{\Box(r\rightarrow\Diamond s)\ po} \\
&\wedge (\neg r_{po} \vee \neg r_{no}) \wedge (\neg s_{po} \vee \neg s_{no}) \wedge (\neg x_{\Box\neg s\ po} \vee \neg x_{\Box\neg s\ no}) \\
&\wedge (\neg x_{\Box(r\rightarrow\Diamond s)\ po} \vee \neg x_{\Box(r\rightarrow\Diamond s)\ no})
\end{aligned}
\tag{8.9}
$$

式 (8.9) の 1–2 行目は式 (8.8) のリテラルを SAT solver 入力用の命題に置き換えたものであり、3–4 行目は、SAT solver 入力用の命題 p_{po}, p_{no} が同時に真とはならないことを表す追加された制約である。また、SAT solver の入力フォーマットに式 (8.9) をエンコードすると図 8.3 のようになる。

この変換後の論理式を入力として用いることで、SAT solver の出力する解に変換元の命題 p に関するドントケアを含めた真偽値についての言及を盛り込むことが可能となる。ドントケアの命題変数を含んだモデルは、その命題変数に真または偽の値を割り当てたモデルのどちらも表すため、真偽値が確定しているモデルに代えてドントケアを含むモデルを SAT 問題の解として用いると、出力すべきモデルの総数を減らすことができる。この論理式変換と、8.4 節で紹介する極小モデル生成手続きを用

```
c r : 1,2, s : 3,4
c []!s : 5,6, [](r -> <>s) : 7,8
c
c !r + s + !s
c !r + s + ![]!s
c [](r -> <>s)
c
c !(r_po) * !(r_no) ...
c
p cnf 8 7
2 3 4 0
2 3 6 0
7 0
-1 -2 0
-3 -4 0
-5 -6 0
-7 -8 0
```

図 8.3: 式 (8.8) を SAT solver 入力フォーマットへエンコードした例

いることにより、ドントケアとなり得る命題変数を多く含んだモデルを手に入れることが可能となる。

8.4 SAT 問題の極小モデル生成手続き

本節では、提案する実装の核の技術として用いる極小モデル生成手続きについて、SAT solver を用いた実装法とともに紹介する。本節で紹介する極小モデル生成手続きは、[46] にて提案されたものである。

SAT solver の出力として得られるモデルは命題変数の真偽値割り当てを表す。このモデルは、そのモデルにおいて真として現れる命題変数だけを集めた命題変数集合を用いて表現することもできる。この命題変数集合を用いて、モデルの大小関係を定義 8.2 のように定め、この順序において極小となるモデルを極小モデルと定める。

定義 8.2 (モデルの大小関係)
P を命題変数集合とし、モデル m_1, m_2 を表す命題変数集合をそれぞれ M_1, M_2 とする。$M_1 \cap P \subset M_2 \cap P$ となるとき、m_1 は m_2 より P に関して小さいという。

定義 8.3 (極小モデル)
φ を命題論理式、P を命題変数集合、m を φ のモデルとする。P に関して m より小さい φ のモデルが存在しない時、m は P に関して極小モデルであるという。

以上のように極小モデルを定義すると、極小モデルと命題論理式の充足可能性に関して次の定理が成立する。

定理 8.3 (極小モデルの性質 [46])
φ を命題論理式、P を命題変数集合、m を φ のモデル、M をモデル m を表す命題変数集合とする。また、$M \cap P = \{a_1, \ldots, a_l\}$, $\overline{M} \cap P = \{b_1, \ldots, b_n\}$ とする。このとき、m が P に関して φ の極小モデルであるならば、次の論理式

$$\varphi \wedge \neg(a_1 \wedge \cdots \wedge a_l) \wedge (\neg b_1 \wedge \cdots \wedge \neg b_n)$$

は充足不能である。逆もまた正しい。

定理 8.3 の性質を利用することで、命題論理式の極小モデルを生成し枚挙する手続きを構成することができる。

手続き 8.1 (極小モデル生成手続き [46])
入力として CNF の形の命題論理式をとり、命題論理式の極小モデルを枚挙する。この手続きは以下のステップで処理を行う。

1: *constMinimalModel*(φ : CNF formula)
2: **while** (isSatisfiable(φ))
3: m := solve(φ); /* モデル m の発見 */
4: α := makeConstraint(m);
5: β := makeAssumption(m);
6: $\varphi := \varphi \wedge \alpha$;
7: **if** ($\neg$ isSatisfiable($\varphi \wedge \beta$))
8: **output**(m); /* m は極小モデル */

ただし、以上の擬似コードにおける isSatisfiable, solve, makeConstraint, makeAssumption は、次のような関数とする。

isSatisfiable 命題論理式を入力とし、その式が充足可能であれば真を、充足不能であれば偽を出力する関数。

solve 命題論理式を入力とし、その式のモデルを一つ出力する関数。

makeConstraint モデルを入力とし命題論理式を出力する関数。出力される式は定理 8.3 の $\neg(a_1 \wedge \cdots \wedge a_l)$ を表す式である。ただし、P は原子命題全体の集合 $\mathcal{P}$ とする。

makeAssumption モデルを入力とし命題論理式を出力する関数。出力される式は定理 8.3 の $\neg b_1 \wedge \cdots \wedge \neg b_n$ を表す式である。ただし、P は原子命題全体の集合 $\mathcal{P}$ とする。

8.5 SAT solver を用いた式の分解手続き

8.3 節で与えた、論理式の変換方法を利用することで、あるリテラル p ($\neg p$) が論理式中に現れないことを $\neg p_{po}$ ($\neg p_{no}$) を用いて表現できる。元の命題変数 p の真偽がドントケアとして扱える場合、SAT solver の解に $\neg p_{po}$, $\neg p_{no}$ がともに現れることとなる。つまり、元の変数にドントケアなものが多く含まれるモデルは、変換後の命題の否定リテラルを多く含むモデルとして表現される。

一方、8.4 節で紹介した極小モデル生成手続きを用いると、SAT solver の出力として否定リテラルを多く含む小さなモデルを獲得できる。モデル中に否定リテラルを多く含めば、二つの命題 p_{po}, p_{no} がともに否定リテラルとして出現する可能性も増える。

そこで、8.3 節の論理式変換法と 8.4 節の極小モデル生成手続きを利用し、SAT solver の出力として否定リテラルを多く含む小さなモデルを獲得することで、ドントケアとして扱える命題変数を多く含むモデルを間接的に獲得でき、結果として小さなプレステートグラフを獲得できる可能性が大きくなることが予想される。以下に、これらを総合した式の分解手続きを示す。

手続き 8.2 (SAT solver を用いた式の分解手続き)
この手続きは式集合 S を入力とし、S の分解結果を表す論理式集合を一つずつ出力する。

1. 式集合 S の各論理式に対して式の分解手続きを適用し、その結果に 8.2 節の時間式変数を用いる変換 $conv$ を適用して、CNF 形式の命題論理式を得る。

2. ステップ 1 で得た CNF 形式の論理式をすべて論理積でつなぎ、8.3 節の論理式変換方法を利用して SAT solver 入力用の論理式に変換する。そして、その結果を手続き 8.1 に適用しそれが停止するまで極小モデルをひとつずつ取得し、そのモデルを式集合の形に逆変換して出力する。

定理 8.4 (手続き 8.2 の正当性)
式集合 S に手続き 8.2 を適用すると、式集合 S に対する式の分解結果をすべて獲得できる。

証明
ステップ 2 で利用する手続き 8.1 は、入力の命題論理式のすべての極小モデルを出力する手続きである。したがって、入力の命題論理式に対するモデルであるにもかかわらず、手続き 8.1 にて出力されなかったモデル m は、手続き 8.1 にて出力されるある極小モデル m_0 よりも大きいモデルである。

一方、ある命題論理式を 8.3 節の方法を用いて変換し、これを SAT solver の入力とした場合、SAT solver の出力するモデルの大小は、変換元のリテラルの出現数の大小に一致する。

したがって、m が表現する変換元の命題変数に対する真偽値割り当ては、m_0 の表現する真偽値割り当てよりも、多くの命題に対して真偽値を割り当てていると言える。

これらの真偽値割り当てが変換元の論理式に対するモデルとなっていることから、m では真偽値が割り当てられているにもかかわらず、m_0 では真偽値が割り当てられていない変換元の命題変数は、ドントケアとして扱える命題変数である。したがって、m_0 の表すモデルは m を含んでいるため、変換元の論理式についてのすべてのモデルを獲得するためには、変換後の m_0 のような極小モデルをすべて獲得すれば十分である。

表 8.2: スタンドアロン実行の実験に利用した計算機環境

CPU	AMD Opteron^{TM}880 2.4GHz × 8
メモリ	32GB
OS	SUSE LINUX Enterprise Server 9
Java	Version 1.5.0_11
SAT4J	Version 2.0.4.v20081103

手続き 8.1 は、変換後のすべての極小モデルを獲得するため、変換元の命題論理式に対して必要なすべての値割り当てが獲得できることとなる。したがって、手続き 8.2 を用いることにより式集合 S に対する式の分解結果をすべて獲得できる。 ■

8.6 実験

本節では、前節までに述べた SAT solver を用いた式の分解手続きの実装手法の効果を二つの実験環境のもとで確認する。

Java 言語と SAT solver 実装として Sat4j [6] を用いて、本章で提案した実装手法により式の分解手続きを実装する。まず、スタンドアロン実行において従来の BDD を利用した実装と、SAT solver を利用した実装について、構成されるグラフの大きさ、グラフ構成にかかる時間および必要となるメモリ空間の大きさを比較する。

次に、6 で実装を行った T^{3}ED の式分解手続き部分を、SAT solver を利用した実装に代え、BDD を利用した実装と、グラフ構成時間について比較を行う。

スタンドアロン実行での実験は、TSUBAME グリッドクラスタ [45] の 1 ノードを利用しソフトウェアの実行を行う。この計算機環境の主な仕様は、表 8.2 の通りである。また、後者の実験の実験環境は、5.4 節で利用した計算機環境を用いる。6.4 節同様、実行環境上には一つのマスタプロセスと、複数のワーカプロセスを用意し、一つの計算機ノードあたりに一つのプロセスを割り当てるようにプログラムを実行する。どちらの実験においても、入力として与える仕様には、巻末の付録に添付した簡易 n 階建てエレベータシステム仕様と、n プロセス排他制御システム仕様を利用する。

表 8.3: スタンドアロン実行で構成されるタブローグラフの大きさ

	ノード数	エッジ数
3 階建てエレベータ	88	1,644
4 階建てエレベータ	599	22,976
5 階建てエレベータ	4,247	329,476
5 プロセス排他制御	32	648
6 プロセス排他制御	64	2,187
7 プロセス排他制御	128	7,290
8 プロセス排他制御	256	24,057
9 プロセス排他制御	512	78,732
10 プロセス排他制御	1,024	255,879
11 プロセス排他制御	2,048	826,686

実験環境としては、5.4 節で利用した計算機環境と入力を用いる。また 6.4 節同様、実行環境上には一つのマスタプロセスと、複数のワーカプロセスを用意し、一つの計算機ノードあたりに一つのプロセスを割り当てるようにプログラムを実行する。

8.6.1 実験結果

スタンドアロン実行での実験結果については、表 8.3–表 8.5 にまとめている。また、T^3ED の式の分解手続き部分を変更しての実験については、その実験結果を表 8.6、表 8.7 にまとめた。

まず、スタンドアロン実行での実験結果について述べる。式の分解手続きに SAT solver を用いた実装（以降、SAT solver 利用実装と表記）と従来の BDD を用いた実装（以降、BDD 利用実装と表記）とで構成されるグラフに違いがあるか確認を行った。その結果、どちらの実装法でも構成されるタブローグラフの大きさに違いはなく、各仕様について構成されるグラフの大きさは表 8.3 の通りであった。

それぞれの仕様を入力とした場合のタブローグラフ構成にかかる時間を計測した結果は、表 8.4 の通りであった。5 階建てエレベータ仕様を入力とした時の BDD 利用実装での実験は、1 時間以内に終了しなかった。実験環境の使用条件により、ひとつのアプリケーションを 1 時間以上動

表 8.4: スタンドアロン実行にてタブローグラフの構成にかかる時間 (ms)

	SAT solver 利用実装	BDD 利用実装
3 階建てエレベータ	7,098	5,535
4 階建てエレベータ	111,247	265,087
5 階建てエレベータ	1,706,285	$>$ 3,600,000
5 プロセス排他制御	876	298
6 プロセス排他制御	2,331	870
7 プロセス排他制御	11,992	5,127
8 プロセス排他制御	41,042	21,402
9 プロセス排他制御	155,044	92,074
10 プロセス排他制御	582,065	490,111
11 プロセス排他制御	$>$ 3,600,000	2,537,162

作させる続けることができないためその時点で実験を終了した。そのため、表 8.4 では “$> 3,600,000$” と表している。同様に、11 プロセス排他制御システム仕様を入力とした時の SAT solver 利用実装に関しても同様である。

また、表 8.5 は、グラフ構成直後のメモリ使用量について計測したものをまとめたものである。

次に、T^{3}ED の式分解手続き部分を変更して行った実験の結果について述べる。表 8.6 は、式分解手続き部分に BDD 利用実装と、SAT solver 利用実装についてそれぞれプレステートグラフの構成にかかった時間をまとめたものである。6.4 節と同様、台数の欄にはワーカプロセスの数を表記している。

それぞれの実験結果について、BDD 利用実装にて N 台のワーカプロセスを利用したときの処理速度を 1 とし、同じ台数のワーカプロセスを用いた SAT solver を利用実装がどの程度速度が向上しているかを次の式を用いて求め、それを表 8.7 にまとめている。

$$\text{速度向上率} = \frac{time_{BDD}(N)}{time_{SAT}(N)}$$

ただし、$time_{BDD}(N)$ は、BDD を利用した実装にて N 台のワーカプロセスを用いてプレステートグラフを構成した際にかかる処理時間、$time_{SAT}(N)$ は、SAT solver を利用した実装にて N 台のワーカプロセスを用いてプレ

表 8.5: スタンドアロン実行でのタブローグラフ構成直後の使用メモリ (byte)

	SAT solver 利用実装	BDD 利用実装
3 階建てエレベータ	3,646,992	26,470,256
4 階建てエレベータ	9,782,600	472,773,992
5 階建てエレベータ	74,160,952	time out
5 プロセス排他制御	2,473,112	3,179,000
6 プロセス排他制御	3,074,240	3,782,768
7 プロセス排他制御	2,877,184	6,080,584
8 プロセス排他制御	8,178,040	10,794,968
9 プロセス排他制御	14,029,936	28,572,296
10 プロセス排他制御	51,258,240	91,396,648
11 プロセス排他制御	time out	143,148,120

ステートグラフを構成した際にかかる処理時間を表す。

8.6.2 評価

前節で述べた通り、今回の実験で用いた仕様では、式の分解手続きの実装として BDD 利用実装、SAT solver 利用実装のどちらを利用しても、構成されるプレステートグラフのサイズに変化はなかった。この結果だけからは、“どちらの実装を用いても構成されるプレステートグラフの形状は全く同じである” とまでは言えないが、“構成されるプレステートグラフの大きさを小さく抑えるという効率化に関して、SAT solver 利用実装は、BDD 利用実装と同程度の能力を持つ” と十分に結論づけられる。これは、ドントケアな命題変数を表現できるエンコードと極小モデル生成手続きを併せて利用したことによってもたらされた効果である。

表 8.4 から、スタンドアロン実行において n プロセス排他制御システム仕様を入力とした実験では、本稿で提案した SAT solver 利用実装の方が 1.2–2.9 倍程度，時間がかかっていることがわかる。しかし、その比率は仕様が複雑になるほど小さく抑えられている。また、n 階建てエレベータ仕様を入力とした実験では、簡易 3 階建てエレベータ仕様では BDD 利用実装の方が勝ったものの、簡易 4 階建て、5 階建てエレベータ仕様では

表 8.6: T^{3}ED の各実装について計測したプレステートグラフ構成時間 (単位:sec)

	3 階建て		4 階建て		5 階建て	
T^{3}E	4.64		57.2		2739	
台数	BDD	SAT-solver	BDD	SAT-solver	BDD	SAT-solver
2	5.10	3.10	58.2	7.52	3212	74.0
3	4.10	3.11	40.4	6.13	1817	53.1
4	3.10	3.12	29.2	5.93	1319	45.0
5	3.10	3.10	22.8	5.53	1091	34.8
6	3.11	3.10	20.2	5.75	938	31.4
7	2.11	2.90	17.2	5.34	800	26.7
8	2.10	3.10	16.2	5.15	694	26.7
9	2.10	3.10	15.2	5.54	609	24.6
10	2.10	3.11	12.8	5.34	541	23.8
	8 プロセス		9 プロセス		10 プロセス	
T^{3}E	7.16		45.0		221.8	
	BDD	SAT-solver	BDD	SAT-solver	BDD	SAT-solver
2	11.91	6.11	54.7	14.2	326.2	35.2
3	9.12	6.11	36.8	9.3	247.0	22.6
4	7.52	4.91	29.2	8.5	167.6	20.3
5	6.11	5.11	23.0	7.5	137.6	15.2
6	7.32	5.72	30.2	9.2	119.8	15.4
7	5.12	4.92	18.4	7.0	91.6	13.4
8	4.31	4.12	27.2	8.6	84.4	15.0
9	4.72	4.11	14.2	6.4	90.5	12.8
10	4.32	4.32	14.8	6.9	68.6	11.9

表 8.7: T^{3}ED の各実装について BDD を利用した実装と比較したときの速度向上率

台数	3 階建て	4 階建て	5 階建て
2	1.64	7.73	43.4
3	1.32	6.58	34.2
4	0.99	4.92	29.3
5	1.00	4.11	31.4
6	1.00	3.51	29.9
7	0.73	3.21	30.0
8	0.68	3.14	26.0
9	0.68	2.74	24.8
10	0.68	2.39	22.8
	8 プロセス	9 プロセス	10 プロセス
2	1.95	3.87	9.26
3	1.49	3.94	10.92
4	1.53	3.41	8.26
5	1.20	3.04	9.04
6	1.28	3.29	7.78
7	1.04	2.64	6.83
8	1.05	3.17	5.62
9	1.15	2.23	7.07
10	1.00	2.13	5.79

SAT solver 利用実装の方がグラフ構成時間が短い。簡易 4 階建てエレベータ仕様については、BDD 利用実装の約 2.4 倍の処理速度を SAT solver 利用実装が達成している。

T^{3}ED における実験では、表 8.6、表 8.7 のとおり、ほとんどの実行について SAT solver 利用実装の方が処理時間が短く高速化されていることが分かる。とくに、簡易 5 階建てエレベータ仕様に対しては、SAT solver 利用実装は BDD 利用実装に比べ 20 倍以上もの処理速度が観測されている。

これらの実験結果から、グラフ構成時間に関しても並列分散化の有無を問わず、SAT solver 利用実装は BDD 利用実装に比べて遜色のない能力を持っていると考える。これは、式の分解手続きの実装に、高度に高速化のチューニングを受けた既存の SAT solver を用いることが、処理速度

向上の面でかなり有利であることを表している。

とくに、T^{3}ED に対する実験においては、BDD 利用実装ではワーカプロセス中の次ノード計算スレッドは、BDD 情報の再利用のため、式の分解手続きで BDD を作るたびに、それをワーカプロセスに登録しておく必要がある。この情報はワーカプロセス中で共通であるため、これを書き換える際には次ノード計算スレッド間で排他処理が必要となってしまう。しかし、SAT solver 利用実装では、BDD 利用実装のように次ノード計算スレッド同士で共有する情報を持たず、それに伴う排他制御も存在しないため、スレッド処理による並列化効果が大きくなる。これも、SAT solver を利用した実装が BDD を利用した実装に比べ高速化された要因の一つと考える。

表 8.5 から、すべてのグラフ情報を一つの計算機で持つ必要のあるスタンドアロン実行において、どの仕様を入力として与えた時にも，SAT solver 利用実装の方がメモリ使用量が小さいということが分かる。とくに，n 階建てエレベータ仕様において，それが顕著に現れている．この結果から、空間的な効率化の観点からみると、BDD 利用実装よりも SAT solver 利用実装の方が勝っていると考える。BDD 利用実装は，一度処理に利用した BDD をすべて保持しておくため，メモリ使用量が SAT solver 利用実装に比べて大きくなるのは当然である．

ここまでの実験結果から、式の分解手続きを SAT solver 利用実装とすることは、BDD 利用実装とすることに比べて遜色なく、とくに T^{3}ED に SAT solver 利用実装を用いることは、グラフ構成時間の短縮に大きく貢献できると考える。

しかし、4 章で与えた段階的充足可能性判定手続きは、BDD 利用実装の式の分解手続きで作られた BDD や、その BDD を変形してつくる $\mathrm{BDD}_{\mathrm{PS}}$ や $\mathrm{BDD}_{\mathrm{MPS}\lrcorner\mathrm{r}}$ を用いる手続きであるため、式の分解手続きで BDD 情報が付加されない SAT solver 利用実装で作成したプレステートグラフはそのままでは利用できない。SAT solver 利用実装を生かすためには、BDD に頼らない段階的充足可能性判定手続きの実装法について考察する必要があり、今後の課題として残る。

ただし、決定化手続き以降の処理を行わない充足可能性判定手続きでは、SAT solver 利用実装が構成するプレステートグラフで十分であり、プレステートグラフの構成時間の短縮は充足可能性判定処理にかかる時間の短縮に直結するため、本章で与えた実装手法は依然として有用であると考える。

以上の実験結果および考察から、本章で与えたSAT solverを利用した式の分解手続きの実装手法は、段階的充足可能性判定手続きの実装に用いることができないという課題は存在するものの、その実装をプレステートグラフ構成に利用すると、構成されるプレステートグラフの大きさはBDD利用実装と同程度に抑えられる、BDD利用実装に比べ並列分散化実装において処理速度の面で大きく優位性をもちプレステートグラフの構成時間を大幅に短縮できるなど、式の分解手続きの実装手法として有用な実装手法であると結論づける。

第9章　まとめ

本論文では、主に線形時相論理により記述されたリアクティブシステム仕様の性質検証器環境を並列分散環境上に実現する手法について述べた。

まず、これまで与えられていた誤判定を起こす可能性のある手続きに代わる、タブロー法による完全な段階的充足可能性判定手続きを与えた。この手続きは、すでに与えられている決定性オートマトンによる段階的充足可能性判定手続きをタブロー法に応用したもので、従来の手続きに内部手続きの一つとしてタブローグラフの決定化手続きを追加したものである。

また特殊な二分決定グラフを 2 種類導入し、これらを用いて段階的充足可能性判定器を実現する方法を与えた。その実現方法では、グラフの決定化手続きと行き止まり判定手続きにそれぞれ異なる種類の二分決定グラフを用いている。決定化手続きで用いる二分決定グラフ $\mathrm{BDD}_{\mathrm{PS}}$ は、終端ノードがプレステートの集合であり、$\mathrm{BDD}_{\mathrm{PS}}$ の和集合演算を用いることで次マクロプレステート集合が計算できる。行き止まり判定手続きで用いる二分決定グラフ $\mathrm{BDD}_{\mathrm{MPS\_r}}$ は、終端ノードがマクロプレステートの集合となっており、各マクロプレステートから作られた $\mathrm{BDD}_{\mathrm{MPS\_r}}$ が空集合を終端ノードとして持つか確認することで、そのマクロプレステートが行き止まりであるかを判定できる。

次に、本論文では仕様の性質検証器環境を並列分散環境上に実現する方法として、MPI を利用する手法と分散オブジェクト技術を利用する手法の大きく分けて二つの実装手法を示し比較した。まず、タブローグラフの構成手続きについて、次ノード集合の作成処理に注目することで並列処理可能であることを示し、MPI および分散オブジェクト技術を利用してこの手続きを実装する手法を提案した。MPI を用いた実装手法では、構成されるタブローグラフは一つのプロセスに一元管理されるのに対して、分散オブジェクト技術を利用した実装では、タブローグラフはその構成中に実行環境全体に分散されるという違いがある。どちらの実装も、グラフ構成処理が並列化されているため、実行環境上に用意する計算機

を増やすことで処理速度を上げることができ、これを実験により確認した。また、グラフ情報を分散して保持する分散オブジェクト技術を利用した実装では、計算機を増やすことで1台のメモリ負担が抑えられることも同時に確認した。

さらに、二分決定グラフを利用するグラフの決定化手続きと行き止まり判定手続きについても分散環境上に実現できることを述べ、実際にこれらの手続きに対し分散オブジェクト技術を利用した実装手法を与え、段階的充足可能性判定器を並列分散環境上に実現できることを示した。グラフの決定化手続きについては、ノードが分散されているプレステートグラフから、そのプレステートグラフ同様、ノードが分散配置されるマクロプレステートグラフを構成する実装手法を与えた。また、行き止まり判定手続きについては、マクロプレステートが分散配置されている点に着目し、各プロセスが自らの保持するマクロプレステートについてそれぞれ独立に処理を行う並列化手法を与えた。

その他、グラフ構成手続きの内部手続きである式の分解手続きの実装手法として、SAT solver を利用する手法を提案した。提案手法では、プレステートに対する式の分解手続きを、そのプレステートに含まれる各論理式についての式分解結果を命題論理式とみなしたときに、それらすべての命題論理式を満足する真偽値割り当てを不足なく列挙することととらえ、この真偽値割り当ての発見にSAT solver を利用する。あわせて、構成されるタブローグラフの大きさを従来の実装と同程度に抑えるため、一つの命題の真偽値を二つの命題変数にて表現するエンコード手法や命題論理の充足可能性問題に対する極小モデル生成手続きも共に用いている。

既存の高度にチューニングされた高速なSAT solver を利用することで従来のBDD を利用して効率化された実装に比べても、遜色のない効率化が実現できていることを実験により確かめた。また、この実験では、分散オブジェクト技術を利用した実装で用いられている式の分解手続きを、SAT solver を利用した実装に代えることでプレステートグラフの構成速度を大幅に向上できるということも観測された。

今後の課題としては、本論文で扱わなかった強充足可能性や段階的強充足可能性判定手続きに対する並列分散環境上への実装法についての考察が残る。本論文で与えたタブローグラフ構成手続きの並列分散化はこれらの問題に対しても利用できると考えるが、それぞれの手続き固有の内部手続きについての並列化手法などは新たに考察する必要がある。

また、本論文で述べた種々の実装手法を組み合わせた実装について考

察することも今後の研究展望として興味深い。例えば、

- グラフ情報を保持するプロセスを複数用意し、それらのプロセスは T^{3}ED と同様に分散オブジェクト技術を用いてグラフ情報を分散保持する。
- 次ノード計算処理は、T^{3}EM と同様に次ノード計算処理を専門に行うプロセスを用意し、そのプロセスが担当する。

といった実装では T^{3}EM と T^{3}ED の長所が生きた実装となる可能性が大きいと考える。他にも、SAT solver を他の部分手続きに適用することも興味深い課題である。

謝辞

本研究を進めるにあたり、終始ご指導頂きました東京工業大学 米崎直樹教授に心から感謝を申し上げます。また、多くの助言を頂いた萩原茂樹助教、並びに米崎研究室の皆様に感謝いたします。最後に、快く計算機環境を提供下さった産業技術総合研究所 関西センター 連携検証施設の計算機管理者の皆様に感謝いたします。

参考文献

[1] Martín Abadi, Leslie Lamport, and Pierre Wolper. Realizable and unrealizable specifications of reactive systems. In *Proceedings of the 16th International Colloquium on Automata, Language and Programming*, Vol. 372 of *Lecture Notes in Computer Science*, pp. 1–17, 1989.

[2] J. R. Abrial, A. Hoare, and Pierre Chapron. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[3] Takenobu Aoshima, Kenji Sakuma, and Naoki Yonezaki. An efficient verification procedure supporting evolution of reactive system specifications. In *Proceedings of the 4th International Workshop on Principles of Software Evolution (IWPSE '01)*, pp. 182–185, 2001.

[4] Jiří Balcárek, Petr Fišer, and Jan Schmidt. Techniques for sat-based constrained test pattern generation. In *Proceedings of the 14th Euromicro Conference on Digital System Designe (DSD 2011)*, pp. 360–366, 2011.

[5] J. Barnat, L. Brim, I. Černà, and P. Šimeček. DiVinE – the distributed verification environment. In *Proceedings of the 4th International Workshop on Parallel and Distributed Methods in Verification (PDMC 05)*, pp. 89–94, 2005.

[6] Daniel Le Berre and Anne Parrain. The Sat4j Library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 7, pp. 59–64, 2010.

[7] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, Vol. 4, pp. 75–97, 2008.

[8] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In *Proceedings of the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS '99)*, Vol. 1579 of *Lecture Notes in Computer Science*, pp. 193–207, 1999.

[9] Roderick Bloem, Roberto Cavada, Ingo Pill, Marco Roveri, and Andrei Tchaltsev. RAT: A tool for the formal analysis of requirements. In *Proceedings of the 19th International Conference on Computer Aided Verification (CAV 2007)*, Vol. 4590 of *Lecture Notes in Computer Science*, pp. 263–267, 2007.

[10] Roderick Bloem, Alessandro Cimatti, Karin Greimel, Georg Hofferek, Robert Könighofer, Marco Roveri, Viktor Schuppan, and Richard Seeber. RATSY – a new requirements analysis tool with synthesis. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV 2010)*, Vol. 6174 of *Lecture Notes in Computer Science*, pp. 425–429, 2010.

[11] Randal E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, Vol. 24, No. 3, pp. 293–318, 1992.

[12] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. NuSMV: A new symbolic model verifier. In *Proceedings of the 11th Conference on Computer Aided Verification (CAV'99)*, Vol. 1633 of *Lecture Notes in Computer Science*, pp. 495–499, 1999.

[13] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Computational challenges in bounded model checking. *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 7, pp. 174–183, 2005.

[14] Intel Corporation. Intel MPI Library. `http://software.intel.com/en-us/articles/intel-mpi-library/`.

[15] Microsoft Corporation. Windows HPC Server 2008. `http://www.microsoft.com/ja-jp/hpc/default.aspx`.

[16] James M. Crawford and Andrew B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI '94)*, Vol. 2, pp. 1092–1097, 1994.

[17] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, Vol. 5, pp. 394–397, 1962.

[18] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003)*, Vol. 2919 of *Lecture Notes in Computer Science*, pp. 502–518, 2004.

[19] E. Allen Emerson and Joseph Y. Halpern. Decision procedures and expressiveness in the temporal logic of branching time. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing (STOC '82)*, pp. 169–180. ACM, 1982.

[20] International Organization for Standardization. *ISO 8807:1989 Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour*. ISO, 1989.

[21] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 2.2*. High Performance Computing Center Stuttgart, 2009.

[22] Object Management Group. Common object request broker architecture (CORBA) specification, version 3.2. `http://www.omg.org/spec/CORBA/3.2/Interfaces/PDF`, 2011.

[23] Gerand J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, Vol. 23, No. 5, pp. 279–295, 1997.

[24] Cliff B. Jones. *Systematic software development using VDM (2nd ed.)*. Prentice Hall PTR, 1990.

[25] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999.

[26] Y. Kesten, Z. Manna, H. McGuire, and A. Pnueli. A decision algorithm for full propositional temporal logic. In *Proceedings of the 5th International Conference on Computer Aided Verification (CAV '93)*, Vol. 697 of *Lecture Notes in Computer Science*, pp. 97–109, 1993.

[27] Argonne National Laboratory. MPICH2 : High-performance and widely portable MPI. `http://www.mcs.anl.gov/research/projects/mpich2`.

[28] Tracy Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, Vol. 11, No. 1, pp. 4–15, 1992.

[29] Flavio Lerda and Riccard Sisto. Distributed-memory model checking with spin. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, pp. 22–39, 1999.

[30] Ryosei Mori and Naoki Yonezaki. Several realizability concepts in reactive objects. *Information Modeling and Knowledge Bases IV*, pp. 407–424, 1993.

[31] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference (DAC '01)*, pp. 530–535, 2001.

[32] Shin Nakajima and Kokichi Futatsugi. An object-oriented modeling method for algebraic specifications in CafeOBJ. In *Proceedings of the 19th international conference on Software engineering*, pp. 34–44, 1997.

[33] Amir Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pp. 46–57, 1977.

[34] Amir Pnueli and Roni Rosner. On the Synthesis of an Asynchronous Reactive Module. In *Proceedings of the 16th International Collo-*

quium on Automata, Languages and Programming (ICALP '89), Vol. 372 of *Lecture Notes in Computer Science*, pp. 652–671, 1989.

[35] Ben Potter, Jane Sinclair, and David Till. *An introduction to formal specification and Z.* Prentice Hall PTR, 1996.

[36] The HP Java Project. mpiJava. `http://www.hpjava.org/index.html`.

[37] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, Vol. C-35, No. 8, pp. 677–691, 1986.

[38] Jussi Rintanen. Planning with sat, admissible heuristics and a*. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI 2011)*, pp. 2015–2020, 2011.

[39] Roni Rosner. *Modular Synthesis of Reactive Systems.* PhD thesis, Weizmann Institute of Science, 1992.

[40] HIRANO Satoshi. HORB: Distributed Execution of Java Programs. In *Proceedings of the International Conference on Worldwide Computing and Its Applications (WWCA'97)*, Vol. 1274 of *Lecture Notes in Computer Science*, pp. 29–42, 1997.

[41] João P. Marques Silva and Karem A. Sakallah. GRASP – A New Search Algorithm for Satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design (ICCAD '96)*, pp. 220–227, 1996.

[42] Jeffrey M. Squyres and Andrew Lumsdaine. A component architecture for LAM/MPI. In *Proceedings of the 10th European PVM/MPI User's Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, Vol. 2840 of *Lecture Notes in Computer Science*, pp. 379–387, 2003.

[43] The Open MPI Project. Open MPI. `http://www.open-mpi.org/`.

[44] Noriaki Yoshiura. Decision procedures for several properties of reactive system specification. In *Proceedings of the Second Mext-NFS-JSPS International Symposium on Software Security – Theories and Systems*, Vol. 3233 of *Lecture Notes in Computer Science*, pp. 154–173, 2004.

[45] 国立大学法人 東京工業大学学術国際情報センター. TSUBAME グリッドクラスタ. `http://www.gsic.titech.ac.jp/`.

[46] 越村三幸, 鍋島英知, 藤田博, 長谷川隆三. 極小モデル生成とジョブショップスケジューリング問題の解決. 日本ソフトウェア科学会 第25回大会論文集, 2008.

[47] 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, pp. 227–245, 2005.

[48] 安藤崇央, 萩原茂樹, 米崎直樹. SAT solver を用いる LTL タブロー構成法とその評価. 情報処理学会論文誌, Vol. 55, No. 2, pp. 909–921, 2014.

[49] 安藤崇央, 宮本佑樹, 萩原茂樹, 米崎直樹. リアクティブシステム仕様に対する段階的充足可能性判定器の分散オブジェクト技術を利用した実装. コンピューターソフトウェア, Vol. 28, No. 4, pp. 262–281, 2011.

[50] 独立行政法人産業技術総合研究所 関西産学官連携センター組込みシステム技術連携研究体. 連携検証施設 さつき. `http://cfv.jp/cvs/service/facility.html`.

[51] 青島武伸. リアクティブシステム仕様の検証系に関する研究. PhD thesis, 東京工業大学大学院情報理工学研究科 計算工学専攻, 2003.

[52] 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, pp. 30–53, 2003.

[53] 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, pp. 52–63, 2005.

[54] 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. PhD thesis, 東京工業大学大学院理工学研究科 情報工学専攻, 1994.

[55] 森亮靖, 友石正彦, 米崎直樹. 時相論理の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, pp. 25–37, 1998.

付録

簡易 n 階建てエレベータシステム仕様

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。

$\mathcal{R} = \{$

$\mathit{LocBtn}(i) \quad (i = 1 \ldots n),$	// i 階の呼出しボタンが押された。
$\mathit{OpenBtn},$	// “開く” ボタンが押された。
$\mathit{CloseBtn}$	// “閉じる” ボタンが押された。

$\}$

$\mathcal{S} = \{$

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

$\}$

$\varphi' = \{$

// リフトはどこかの階にある。

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

// リフトがある階にある場合、他の階にはない。
$$\Box\left(\bigwedge_{1\le i\le n}\left(Loc(i)\rightarrow\bigwedge_{j=1..n,i\neq j}\neg Loc(j)\right)\right),$$

// 途中階を通る（$n\geq 3$ の場合）。
$$\Box\left(\bigwedge_{1\le i\le n-2}\left(Loc(i)\rightarrow\langle Loc(i+2)\rangle Loc(i+1)\right)\right),$$
$$\Box\left(\bigwedge_{3\le i\le n}\left(Loc(i)\rightarrow\langle Loc(i-2)\rangle Loc(i-1)\right)\right),$$

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

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

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

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

// 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。
$\Box(Open\rightarrow\Diamond OpenTimedOut)$,

// 時間制限内に “開く” ボタンが押されれ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、“閉じる” ボタンが押されればドアを閉じる。
$\Box(CloseBtn \wedge \neg ReqOpen \rightarrow \neg Open)$,

//ドア開放要求があり、可動状態でないならばドアを開く。
$\Box(ReqOpen \wedge \neg Movable \rightarrow Open)$
}

n プロセス排他制御システム仕様

n プロセス排他制御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。

$\mathcal{R} = \{$
// プロセス i がクリティカル領域への進入を要求。
$r_i \quad (1 \leq i \leq n)$
}

$\mathcal{S} = \{$
// プロセス i がクリティカル領域に進入中である。
$c_i \quad (1 \leq i \leq n)$
}

$\varphi' = \{$
// 進入要求が出されれば、いつかそのプロセスはクリティカル領域に進入する。
$\Box \left(\bigwedge_{1 \leq i \leq n} (r_i \rightarrow \Diamond c_i) \right)$,

// 同時に 2 つ以上のプロセスがクリティカル領域に進入することはない。

$$\Box\left(\bigwedge_{1\leq i\leq n}\left(\bigwedge_{i+1\leq j\leq n}\neg(c_i\wedge c_j)\right)\right)$$

}