自己安定分散システムの検証システム

角川裕次[†] 山下悟史[††]

自己安定分散システムとは分散システムの一種で，その初期状況にかかわらず正常な状況に到達するシステムである．自己安定システムは，任意の種類のそして任意の有限数の一時故障に耐えうるシステムである．なぜなら，一時故障が終了した直後の状況を新たな初期状況と考えれば，再び正常なネットワーク状況に回復するからである．自己安定分散システムを設計する際には，その正しさの検証の困難さが問題となる．分散システムの非同期性に加えて，任意の初期状況から正常な状況への到達性の検証が必要だからである．自己安定分散システムの基礎理論の研究の道具として，我々は自動検証システムを開発した．ネットワーク形状やサイズの変更が簡便に行える独自の言語を用いて，自己安定分散システムを記述する．そして本システムは既存のモデル検証系 Spin を利用し，誤りのある自己安定分散システムの場合，正常なネットワーク状況に回復できない実行系列を見つけ出し「反例」として提示することができる．本論文では，開発したシステムの概要を報告する．

A Verification System for Self-Stabilizing Distributed Systems

Hirotsugu Kakugawa[†] and Satoshi Yamashita[††]

Self-stabilizing distributed systems are a class of distributed systems which converge to a correct system state even if they start from arbitrary system states. Self-stabilizing systems can recover from any finite number of transient faults (e.g., message loss, memory corruption). Therefore, they are considered as fault-tolerant distributed systems. When we design a self-stabilizing system, its verification, such as convergence from an arbitrary initial system state to a correct system state, is a difficult task; typically, its proof is long, complex and error-prone. We developed a verification system, based on a model checker Spin, with visualization feature as a tool of self-stabilizing system study. For an incorrect self-stabilizing system, our system displays a counterexample which consists of an initial system state and an execution sequence which does not converge to a correct system state. Our system uses own description language of self-stabilizing distributed systems of arbitrary network topology and size. In this paper, we report outline of our system.

1. はじめに

分散システムとは，プロセスの集合とそれらを結ぶ通信リンクの集合から成るシステムである．任意の状況（configuration）から実行を開始してもいつかは必ず正常な状況に遷移する場合，分散システムは**自己安定**（self-stabilizing）であるという[7),21)]．たとえばメッセージ消失などの一時故障が発生した直後のシステムの状況は，正常ではない状況と考えられる．この状況を初期状況と考えれば，自己安定の性質により，再び正常な状況に回復することが保証されている．したがって自己安定分散システムは，任意の有限数の，任意の種類の一時故障に耐えることができる．このため耐故障性のある分散アルゴリズムの基礎理論の 1 つとして，近年活発に研究されてきている．本論文では，分散アルゴリズムの基礎理論の研究の立場より，アルゴリズムの検証の手法を考察する．

一般的に，自己安定分散システムの正しさの検証は容易ではない．困難さの原因は，以下の理由による．

- どのような初期状況に対しても，正常な状況への遷移を保証しなければならない．
- どのようなプロセスの実行順序の組合せに対しても，正常な状況への遷移を保証しなければならない．

自己安定分散システムの正しさは，文献 18) で提案された variant 関数法を用いることが多い．variant 関数 f は，システムの状況から非負の整数への写像である．variant 関数法による自己安定分散システムの検証は，以下の条件を満たす関数 f を見つけることである．(1) システム状況 γ が正常であるときおよびその

† 広島大学大学院工学研究科情報工学専攻
Department of Information Engineering, Graduate School of Engineering, Hiroshima University
†† 三菱電機インフォメーションシステムズ株式会社
Mitsubishi Electric Information Systems Corporation

ときのみに限り $f(\gamma) = 0$ である．(2) γ を任意の状況とし，γ' を γ の次状況とするとき，$f(\gamma) \geq f(\gamma')$ が成立する．(3) プロセスを任意の順序で実行しても，状況が非正常であれば f の値はいずれは必ず減少する．

この手法を適用するには適切な variant 関数を見つけなければならないが，場合によっては困難な場合がある．この手法を使って証明された自己安定分散システムには，たとえば文献 16)，24) がある．

このほかに，いくつかの検証方法が提案されている．

複雑な自己安定分散システムは，いくつかの基本的な自己安定分散システムを合成することで構成することもできる．たとえば，任意形状のネットワーク上で生成木を構成するシステムと，生成木上でプロセスのリセットをするシステムを合成することで，任意形状のネットワーク上でプロセスのリセットをするシステムが得られる[1]．これまでにいくつかの合成の手法が提案されている（たとえば文献 4)，8)，9) など）．この手法の場合，自己安定分散システムの証明は合成される部分ごとの証明に分解され，その複雑さは減少する．しかし合成される部分の証明は，variant 関数法などの別の手法が必要である．

他の検証法として，状態遷移規則に関する推論に基づく手法がある．文献 3) において Beauquier らは，項書換えシステムに基づいた検証を行っている．この証明手法は，正しいと分かっているシステムに対して適用可能であるが，誤りのあるシステムでの誤りの発見には使うことができない欠点がある．さらに，リングネットワークや線形ネットワークのみが対象となっている．

文献 19) では，Lakhnech らにより線形時相論理に基づいた検証法が提案されている．文献 14) では，Hsu らにより有限状態機械モデルに基づいた検証法が提案されており，例として自己安定極大マッチングアルゴリズム[15] の検証を行っている（文献 15) では，検証には variant 関数法を用いている）．

以上の手法はいずれも，対象とするシステムの振舞いを人間が解釈し，検証のための導出などを手で作業しなくてはならないという欠点がある．

これらの手法とは正反対に，対象システムを記述言語で表現し，システムの動きを機械的に模倣することで検証する手法（モデル検証[6),17)]）も考えられる．文献 22) で Shukla らは，この方式での検証システムを提案している．彼らのシステムでは，プロセスの実行順序の組合せすべてに対してシステムの正当性を検証する．しかしシステムの初期状況をランダムに 1 つ選ぶだけなので，自己安定分散システムの検証としては不完全である．彼らのシステムは，モデル検証系 Spin[10),11),26)] の前処理系として実装されている．

我々は文献 30) で自己安定分散システムの検証を，文献 28)，29) では自己安定分散アルゴリズムのシミュレーションを可視化するシステムを開発した．我々はこれらのシステムを統合し，自己安定分散システムの検証を可視化する環境を構築した．

本論文で報告する我々の検証システムでは，すべての自己安定分散システムの状況とプロセスの実行順序の組合せすべてを検証する．このような検証法は完全に自動的な検証が可能であるという大きな利点があるものの，多くのメモリと長い実行時間を要することは容易に想像できる．モデル検証系 Spin[10),11),26)] には，状態圧縮法[13] や partial order reduction 法[12)☆] など，メモリや実行の効率を高める技法が導入されているので，我々は Spin を利用することとした．

関連した研究として，文献 27) では，モデル検証系 SMV を用いた自己安定アルゴリズムの検証方法が示されている（モデル検証による自己安定システムの検証は本質的に全解探索であり，文献 27) でも同様である）．検証系への入力（アルゴリズムの記述）は，プロセス数やネットワーク形状に依存している場合が多い．つまり，それらを変更するには検証系への入力を大幅に変更する必要がある．

分散アルゴリズム理論の研究では，特定のネットワークのインスタンスのみを取り扱うことは少なく，プロセス数やプロセス間の隣接関係をパラメータ化してアルゴリズムを記述することが多い（たとえば文献 5)，20)，25) など）．一方，汎用の手続き型並列/分散プログラミング言語では[2]，通信相手を直接記述してメッセージ伝達をするか，RPC やランデブなどの抽象化を導入している．しかしネットワーク形状をパラメータ化した記述には注意が払われていないように見受けられる．

そこで本研究では，あらたに記述言語 Spr を導入し，ネットワークの形状やプロセス数をパラメータ化してアルゴリズム本体の記述と分離させている．このため，プロセス数やネットワーク形状をいろいろ変更して検証することが容易となる．

本論文の構成は以下のとおりである．2 章では，自己安定分散システムの定義について簡単に述べる．3

☆ 非同期システムにおいて同時に実行可能な複数の事象を異なる実行順序で実行させても同一の結果となることが分かれば，それらの実行順序の組合せは考えなくて済み，検査すべき実行順序の組合せ数を削減できる．そのような技法を partial order reduction という．

章では，自己安定分散システムの記述言語 Spr を提案する．4 章で我々の検証アルゴリズムを示し，5 章で検証システムと検証例を示す．最後に 6 章では本論文のまとめを行い，今後の課題について述べる．

2. 自己安定分散システム

本章では，本論文で仮定する自己安定分散システムの形式的定義を手短に説明する．詳しくは，たとえば文献 7)，21) を参照されたい．

分散システム S は n 個のプロセス $P_0, P_1, \ldots, P_{n-1}$ の集合より構成される．各プロセスは局所状態を有する．プロセス P_i はその隣接プロセスの局所状態を参照できる．プロセス間の隣接関係は，システムのネットワーク形状によって定義される．本論文では，ネットワーク形状は無向グラフで与えられるものとする．つまり，P_i が P_j に隣接することと，P_j が P_i と隣接することは同値である．

プロセス P_i のアルゴリズムは，以下の形のガード付きコマンドの集合で記述する．

$$
\begin{aligned}
&* [\\
&\quad g_1^i(q_i, q_1^i, \cdots, q_k^i) \rightarrow q_i := f_1^i(q_i, q_1^i, \cdots, q_k^i)\\
&\quad \Box\ g_2^i(q_i, q_1^i, \cdots, q_k^i) \rightarrow q_i := f_2^i(q_i, q_1^i, \cdots, q_k^i)\\
&\quad \Box\ g_3^i(q_i, q_1^i, \cdots, q_k^i) \rightarrow q_i := f_3^i(q_i, q_1^i, \cdots, q_k^i)\\
&\qquad \cdots\\
&]
\end{aligned}
$$

ただし，

- q_i はプロセス P_i の局所状態（局所変数の組）である．
- q_j^i はプロセス P_j^i の局所状態で，P_j^i は P_i の隣接プロセスである．
- g_j^i は**ガード**と呼ばれ，P_i とその隣接プロセスの局所状態に対するブール関数である．
- $q_i := f_j^i(\cdots)$ は**コマンド**と呼ばれ，P_i とその隣接プロセスの局所状態に基づいて P_i の次の局所状態を決定する．

である．

以下の動作の繰返しによって，システムが実行される．

(1) 各プロセスのガードを評価する．真のガードがあるときおよびそのときのみに限り，プロセスは**特権**を持つという．

(2) スケジューラは，特権を持つプロセスの中から 1 つを任意に選ぶ．このスケジューラは，**C デーモン**と呼ばれている[7]．

(3) スケジューラによって選ばれたプロセスは，真であるガードに対応するコマンドを実行する．

システム S の**状況**（configuration）は，すべてのプロセスの局所状態の組で表される．Γ をすべての状況の集合とする．システム S が $\Lambda \subseteq \Gamma$ に関して**自己安定**（self-stabilizing）であるとは，どのような初期状況と C デーモンによる実行に対しても，以下の条件が成り立つときおよびそのときのみをいう．

- いつかは必ず集合 Λ 内の状況へ遷移する．状況 $\lambda \in \Lambda$ は**正当な状況**（legitimate configuration）と呼ばれ，Λ は**正当な状況の集合**と呼ばれる．
- 正当な状況の集合 Λ は，実行に関して閉じている．つまり，どの $\lambda \in \Lambda$ に対しても，もし次状況 λ' が定義されていれば，$\lambda' \in \Lambda$ である．

正当な状況において特権を持つプロセスが 1 つも存在しなければ，その自己安定システムは silent であると呼ばれる．本論文では，silent なシステムを対象とする．つまり，正当な状況になれば実行可能なプロセスは存在しなくなり，動作が停止するようなシステムについて取り扱う．たとえば相互排除を行うシステムは silent ではないが，リーダー選出，ノードの彩色，生成木の構成など，多くの有用な silent な自己安定分散システムが存在する．

3. 記述言語 Spr

自己安定分散システムの自動的な検証のための記述言語として，Spr を提案する．新たな言語を提案する理由は，自己安定分散システムの高水準な記述を可能とすることにある．すでに存在するプログラミング言語を用いて検証のためのプログラムを書く場合，ネットワーク構造を陽に記述する必要がある．そのためネットワーク形状やプロセス数を変えて検証するには，検証プログラムを手作業で変更しなくてはならない．

Spr では，ネットワークの形状やプロセス数をパラメータ化し，アルゴリズムの記述とは分離している．たとえば，「隣接プロセスの中に条件 F が真であるものが存在する」という述語や，「各隣接プロセスの局所変数 v_i の総和」といった関数を用いたアルゴリズムの記述を可能としている．

現在のところ Spr で取り扱えるデータ型は整数に限定している．ブール値の表現には，整数値 0 で偽を表し，それ以外の整数値で真を表している．各プロセスには識別子が付けられており，整数によって識別子を表す．なお，Spr は Lisp 風の記法を用いている．

自己安定分散システム研究において用いられる計算モデルにはさまざまなものがあるが，Spr では以下のモデルを取り扱う．

- 通信方式：状態通信モデル

- 通信リンク：双方向リンク
- 実行：C デーモン
- アルゴリズムの種類：silent
- プロセス識別子：大局的に一意

他のモデル（レジスタ通信/メッセージ通信，単方向リンク，D デーモン/RW デーモン☆，non-silent など）は対象外とし，モデルを限定することで Spr の仕様を特化している．

検証の対象とするシステムの記述は，以下の記述子を用いる．

- **プロセス数の指定：**
  (`the-number-of-processes` n)
- **プロセス識別子の最小値：**
  (`process-id-base` n)
  もしこの値が与えられなければ，1 が用いられる．
- **ネットワーク形状：**
  (`network-topology` ⟨*Name*⟩ [⟨*opt*⟩])
  現在指定可能な形状名⟨*Name*⟩は，以下のとおりである．`linear`（線形），`binary-tree`（二進木），`tree`（木，追加パラメータで子ノード数を指定），`bidirectional-ring`（双方向リング），`unidirectional-ring`（単方向リング），`bipartite`（二部グラフ，追加パラメータで次数を指定），`regular`（正則グラフ，追加パラメータで次数を指定），`complete`（完全グラフ）．一般に形状名は同じでも異なるネットワークが存在するが，Spr では与えられた形状名に応じてある一定の規則でネットワークを生成している☆☆．その具体的な規則については省略する．
- **プロセスの局所変数：**
  (`process-state`
  (⟨var_1⟩ ⟨min_1⟩ ⟨max_1⟩)
  (⟨var_2⟩ ⟨min_2⟩ ⟨max_2⟩)
  ···)
  プロセスの局所変数として，値域が⟨min_i⟩ . . . ⟨max_i⟩である整数変数⟨var_i⟩を宣言する．
- **マクロ定義：**
  (`define-macro` (⟨*Name*⟩ [⟨*Arg*⟩. . .]) ⟨*Body*⟩)
  以下で説明するガード付きコマンドおよび正当な状況の記述の中の出現(⟨*Name*⟩ ⟨*Arg*⟩. . .) を⟨*Body*⟩に置換するようパラメータ付きのマクロを定義する．
- **ガード付きコマンドの集合：**
  (`algorithm` ⟨*Process*⟩
  (⟨$Guard_1$⟩ `->` ⟨$Command_1$⟩)
  (⟨$Guard_2$⟩ `->` ⟨$Command_2$⟩)
  ···)
  ⟨*Process*⟩ ⟶ `all` | `root` | `other` | i
  ⟨*Process*⟩にプロセス識別子を与えることで，どのプロセスのガード付きコマンドの記述かを指定する．ただし`all` の場合，すべてのプロセスが同一のガード付きコマンドを使用する．`root` の場合，根プロセス（最小のプロセス識別子を持つプロセスを根プロセスと呼ぶ）のガード付きコマンドの指定となる．`other` の場合は，それまでにガード付きコマンドが与えられてないプロセスすべてへの指定となる．
  ⟨*Guard*⟩と⟨*Expr*⟩および⟨*Command*⟩に関する構文規則を，それぞれ**図 1** と**図 2** に示す．
- **正当な状況の記述：**
  (`legitimate-state` ⟨*Expr*⟩)
  ⟨*Expr*⟩はプロセスの局所状態に関する述語であり，状況が正当であるときおよびそのときのみに限り真であるとする．

Huang によって提案された自己安定リーダー選出アルゴリズム[16]を，**図 3** に示す．このアルゴリズムでは，プロセス数が素数個の双方向リングネットワークを仮定している．このアルゴリズムを Spr で記述し，プロセス数を 7 とした場合のものを**図 4** に示す．プロセス数を変更するには，1 カ所を修正するだけで済むことが分かる．

図 5 には，Sur らによって提案された二部グラフに対する 2–彩色自己安定アルゴリズム[24]を Spr で記述したものを示す．なおネットワーク形状は，二部グラフを指定している．

現在の仕様では必ずしもすべてのアルゴリズムが記述可能というわけではない．たとえば任意のパラメータ k (≥ 1) に対してある与えられた式の値が k 番目である隣接プロセスを知ることはできない．汎用のプログラミング言語のような形での記述を可能とすることも考えられるが，記述可能ということと記述がコンパクトで理解しやすいということは必ずしも一致しな

☆ D デーモンとは，特権を持ったプロセスのうち任意の 1 つ以上のプロセスを選んで同時に実行させるスケジューラである．RW（Read-Write）デーモンとは，レジスタの読み出し，書き込みそれぞれを不可分な実行単位として，より細かな粒度で実行をスケジュールするものである．

☆☆ たとえば大きさ n のリングネットワークの識別子の付け方は $(n-1)!$ 通りもあるので，すべての組合せを調べるのは事実上不可能である．またこれまでの経験でアルゴリズムの誤りが表面化する場合は，識別子の付け方がランダムかどうかにはあまり関係ない場合が多い．そのため，決定性の規則を用いることとした．

⟨*Guard*⟩ ⟶ ⟨*Expr*⟩

⟨*Expr*⟩ ⟶ `(not` ⟨*Expr*⟩`)` | `(and` ⟨*Expr*$_1$⟩ ...`)` | `(or` ⟨*Expr*$_1$⟩ ...`)` | `(=` ⟨*Expr*$_1$⟩ ⟨*Expr*$_2$⟩`)`
| `(<` ⟨*Expr*$_1$⟩ ⟨*Expr*$_2$⟩`)` | `(<=` ⟨*Expr*$_1$⟩ ⟨*Expr*$_2$⟩`)` | `(>` ⟨*Expr*$_1$⟩ ⟨*Expr*$_2$⟩`)` | `(>=` ⟨*Expr*$_1$⟩ ⟨*Expr*$_2$⟩`)`
| `(+` ⟨*Expr*$_1$⟩ ...`)` | `(-` ⟨*Expr*$_1$⟩ ...`)` | `(*` ⟨*Expr*$_1$⟩ ...`)` | `(/` ⟨*Expr*$_1$⟩ ⟨*Expr*$_2$⟩`)` | `(modulo` ⟨*Expr*$_1$⟩ ⟨*Expr*$_2$⟩`)`
| `(modulo-n-processes` ⟨*Expr*⟩`)` — プロセス数を法とする⟨*Expr*⟩の値.
| `(cond-expr` ⟨*Expr*$_1$⟩ ⟨*Expr*$_2$⟩ ⟨*Expr*$_3$⟩`)` — ⟨*Expr*$_1$⟩が真のとき⟨*Expr*$_2$⟩，それ以外では⟨*Expr*$_3$⟩.
| `(state-ref` ⟨*var*⟩`)` — 局所変数⟨*var*⟩の参照.
| `(state-ref` ⟨*var*⟩ ⟨*Expr*⟩ `)` — プロセス⟨*Expr*⟩の局所変数⟨*var*⟩の参照.
| `(root)` — 根プロセスの識別子. | `(me)` — 自己のプロセス識別子.
| `(right-process)` | `(left-process)` — 右／左に接するプロセス識別子.（リングネットワーク）
| `(itself)` — 条件を満たす隣接プロセスの識別子. `for-each-process`, `for-each-neighbor` 等の内側で使用される.
| `(neighbor?` ⟨*Expr*⟩`)` — ⟨*Expr*⟩ が隣接プロセスの識別子であれば真.
| `(the-neighbor)`
— 条件を満たす隣接プロセスの識別子.⟨*Command*⟩での`let-neighbor` の内側で使用される.⟨*Command*⟩を見よ.
| `(exists-process` ⟨*Expr*⟩`)` | `(exists-neighbor` ⟨*Expr*⟩`)`
— ⟨*Expr*⟩ が真であるような（隣接）プロセスが存在すれば真.
| `(for-each-process` ⟨*Expr*⟩`)` | `(for-each-neighbor` ⟨*Expr*⟩`)`
— 全ての（隣接）プロセスで⟨*Expr*⟩ が真であれば真.
| `(the-number-of-neighbors` [⟨*Expr*⟩] `)` | `(the-number-of-processes` [⟨*Expr*⟩] `)`
— ⟨*Expr*⟩が真であるような（隣接）プロセスの数.⟨*Expr*⟩がない場合は，全ての（隣接）プロセスの数.
| `(neighbor-with-max-id` ⟨*Expr*⟩`)` | `(neighbor-with-min-id` ⟨*Expr*⟩`)`
— ⟨*Expr*⟩ が真であるような隣接プロセスの内，最大（最小）のプロセス識別子.
| `(neighbor-with-max-value` ⟨*Expr*⟩`)` | `(neighbor-with-min-value` ⟨*Expr*⟩`)`
— 隣接プロセスの内で，⟨*Expr*⟩の値が最大（最小）であるプロセスの識別子.
| `(max-value-among-neighbors` ⟨*Expr*⟩`)` | `(min-value-among-neighbors` ⟨*Expr*⟩`)`
— 隣接プロセスの内で，⟨*Expr*⟩の最大（最小）値.
| `(sum-for-each-neighbor` ⟨*Expr*⟩`)` | `(product-for-each-neighbor` ⟨*Expr*⟩`)`
— 各隣接プロセスの⟨*Expr*⟩の値の和（積）.

図 1 ⟨*Guard*⟩ と ⟨*Expr*⟩ に対する構文規則
Fig. 1 Syntax for ⟨*Guard*⟩ and ⟨*Expr*⟩.

⟨*Command*⟩ ⟶
`(begin` ⟨*Command*$_1$⟩ ⟨*Command*$_2$⟩ ...`)` — ⟨*Command*$_1$⟩ ... を逐次実行する.
| `(state-set!` ⟨*variable*⟩ ⟨*Expr*⟩`)` — 局所変数 ⟨*variable*⟩ の値を ⟨*Expr*⟩ にする.
| `(let-neighbor` ⟨*Expr*⟩ ⟨*Command*$_1$⟩ ⟨*Command*$_2$⟩ ...`)`
— ⟨*Expr*⟩が真であるプロセスの識別子を`(the-neighbor)` の値とし、⟨*Command*$_1$⟩ ... を実行.

図 2 ⟨*Command*⟩に対する構文規則
Fig. 2 Syntax for ⟨*Command*⟩.

いので，現在のところ汎用のプログラミング言語のような記述機能は導入していない．しかし図 3 と図 4 を比べれば分かるように，Spr では与えられたアルゴリズムと正当な状況を直接的に記述できる，という利点があげられる．

4. 検 証 法

この章では，自己安定システム S を検証する手法を提案する．S のプロセス数を n とする．説明を単純にするため，プロセスには 0 から $n-1$ までの番号が付いているものとする．プロセス P_i の局所変数を $v_1^i, v_2^i, \ldots, v_m^i$ とし，R_j を変数 v_j^i の値域とする（変

数の値域はどのプロセスでも同じである）．g_j^i をプロセス P_i の j 番目のガードとし，L を状況が正当か否かを表す述語とする．

G をすべてのプロセスのすべてのガードの論理和，つまり $G = \vee_{i,j} g_j^i$ とする．よって $G(\gamma)$ が真であるときおよびそのときのみに限り，特権を持つプロセスが存在する．

我々の提案する検証アルゴリズムは，**図 6** のとおりである．なお，S のすべての状況の集合 Γ_S は以下のように定義される．

$$\Gamma_S = \{(v_1^0, ..., v_m^0, v_1^1, ..., v_m^1, ..., v_1^{n-1}, ..., v_m^{n-1}) \mid \forall i, j[v_j^i \in R_j]\}$$

手続き Verify(S) により，S の検証が行われる．S の各状況を初期状況として，Traverse を呼び出す．手続き Traverse(γ, *path*) は，γ を初期状況とし到達可能なすべての状況を探索する．すでに探索が済んだ状況を再び探索しないよう，訪問した状況は変数 *Visited* に蓄えておく．引数 *path* には，初期状況から現在の状況までの実行履歴を保持している．正常な状況への到達が永久に不可能な無限ループが存在するか否かは，この値を調べる．

任意の状況 γ_0 に対し γ_0 を初期状況とする任意の実行を考える．状況 γ を実行過程での任意の状況とす

プロセス: $P_0, P_1, ..., P_{n-1}$

局所変数: $label_i \in \{0, 1, ..., (n-1)\}$

マクロ定義:

$$g(a,b) = \begin{cases} n & (a = b) \\ b - a (\mathrm{mod} n) & \text{otherwise} \end{cases}$$

$$x = g(label_{i-1}, label_i)$$

$$y = g(label_i, label_{i+1})$$

アルゴリズム:

$$* [$$

$$(x = y) \wedge (y = n) \rightarrow label_i = label_i + 1 (\mathrm{mod} n)$$

$$\Box\ (x < y) \rightarrow label_i = label_i + 1 (\mathrm{mod} n)$$

$$]$$

図 3 自己安定リーダー選出アルゴリズム[16)]
Fig. 3 A self-stabilizing leader election algorithm[16)].

```
(the-number-of-processes  7)
(process-id-base  0)
(network-topology  bidirectional-ring)
(process-state  (label 0 (- (the-number-of-processes) 1)))
(define-macro  (g a b)
    (cond-expr (= a b) (the-number-of-processes) (modulo-n-processes (- b a))))
(define-macro  (x)  (g (state-ref label (left-process)) (state-ref label)))
(define-macro  (y)  (g (state-ref label) (state-ref label (right-process))))

(algorithm  all
  ((and (= (x) (y)) (= (y) (the-number-of-processes)))    ;; *** Rule 1
   -> (state-set! label (modulo-n-processes (+ (state-ref label) 1))))
  ((< (x) (y))                                            ;; *** Rule 2
   -> (state-set! label (modulo-n-processes (+ (state-ref label) 1)))))

(legitimate-state
  (and (for-each-process (= (x) (y)))
       (= 1 (the-number-of-processes (= (state-ref label) 0)))))
```

図 4 Spr による自己安定リーダー選出アルゴリズム[16)]
Fig. 4 A self-stabilizing leader election algorithm[16)] in Spr.

```
(the-number-of-processes  7)
(process-id-base  1)
(network-topology  bipartite 2)
(process-state  (level 0 (the-number-of-processes)))

(algorithm  root
  ((!= (state-ref level) 0)
   -> (state-set! level 0)) )

(algorithm  other
  ((and (!= (state-ref level (neighbor-with-min-value (state-ref level)))
            (- (the-number-of-processes) 1))
        (!= (state-ref level)
            (+ 1 (state-ref level (neighbor-with-min-value (state-ref level))))))
   -> (state-set! level
        (+ 1 (state-ref level (neighbor-with-min-value (state-ref level)))))))

(legitimate-state
  (and (= (state-ref level (root)) 0)
       (for-each-non-root-process
        (= (state-ref level)
           (+ 1 (state-ref level (neighbor-with-min-value (state-ref level))))))))
```

図 5 Spr での自己安定彩色アルゴリズム[24)]
Fig. 5 A self-stabilizing coloring algorithm[24)] in Spr.

```
var Visit : subset of Γ_S;

Verify(S) {
  Visit := ∅;
  for each γ ∈ Γ_S
    Traverse(γ, ε);
  /* S is verified */
}

Traverse(γ, path) {
  if (γ appears in path)
    abort;    /* infinite loop exists */
  if (γ ∈ Visit)
    return;
  Visit := Visit ∪ {γ};
  if (G(γ)) {
    if (L(γ))
      abort;    /* legitimate but non-stable */
    for each γ' reachable by single step from γ
      Traverse(γ', path · γ);
  } else {
    if (¬L(γ))
      abort;    /* stable but non-legitimate */
  }
}
```

図 6 検証アルゴリズム
Fig. 6 A verification algorithm.

るとき，もし以下の条件の 1 つでも成立すれば S は正しくないシステムである．

- $G(\gamma) \wedge L(\gamma)$
  S が silent でないことを検出している．
- $\neg G(\gamma) \wedge \neg L(\gamma)$
  γ からは正常な状況に到達不可能であることを検出している．
- γ が *path* に含まれる
  γ_0 を初期状況，$path = \gamma_0 \cdot \gamma_1 \cdots \gamma_m$，$\gamma_m = \gamma_i$ $(0 \leq i \leq m)$ とする．γ_0 を初期状況とする実行 $\gamma_0, \gamma_1, \ldots \gamma_i, \gamma_{i+1}, \ldots, \gamma_m, \gamma_{i+1}, \ldots$ は，正当な状況に永久に遷移できない．

ここで検出した γ あるいは *path* は，S の反例となる．

本論文では silent な自己安定分散システムのみを取り扱っているが，上記のアルゴリズムを非 silent な場合に変更するのも容易である．紙面の都合上，説明は省略する．

なおこの検証方法では，特定のネットワーク形状（プロセス数や識別子の付け方も含む）に対してのみ検証を行うことになる．つまり，あるネットワーク形状のクラスすべてのインスタンスに対する検証ではない．しかしこれまでの経験では，特定のインスタンスにおいてのみ誤り（初期状況と安定しない無限の実行列の存在）が表面化することはあまりなく，もし誤りがあれば多くのインスタンスにおいて表面化する場合がほとんどである．そのため，特定のインスタンスに対する検証であっても本手法はアルゴリズムの誤りの発見に十分有効と考える．

```
int s_v1[n];      /* 各プロセス i の局所変数 s_v1 */
int s_v2[n];      /* 各プロセス i の局所変数 s_v2 */
  ...
proctype ss_alg_1() {    /* プロセス 1 */
  do  :: atomic { g_1^1 → c_1^1; }    /* コマンド実行 */
      :: atomic { g_2^1 → c_2^1; }
         ...
  od
}
  ⋮
proctype ss_alg_n() {    /* プロセス n */
  do  :: atomic { g_1^n → c_1^n; }
      :: atomic { g_2^n → c_2^n; }
         ...
  od
}
proctype monitor() {    /* チェッカ */
  if  :: G ∧ L → assert(0);
      :: ¬G ∧ ¬L → assert(0);
      :: else → skip;
  fi
}
init {    /* 初期化 */
  各プロセス i の局所変数 s_v_j[i] の値を非決定的に設
  定する．変数の値域を α_j..β_j とすると、以下の通り．
      s_v_j[i] = α;
      do  :: (s_v_j[i] ≤ β) → break;
          :: (s_v_j[i] < β) → s_v_j[i]++;
      od;
  atomic {    /* プロセスとチェッカを生成 */
      run ss_alg_1();  ···  run ss_alg_n();
      run monitor();
  }
}
```

図 7 出力コード（Promela 言語）の概要
Fig. 7 Outline of generated code in Promela.

5. 検証システムと検証例

図 6 の検証アルゴリズムを実現する，Spin[10),11),26)] の前処理系を実装した．Spin は入力モデル記述に Promela と呼ばれる言語を採用している．本前処理系は，Spr 言語から Promela 言語へのコンパイラである．コンパイラはプログラミング言語 Scheme で記述した．**図 7** に，出力される Promela コードの概要を示す．この Promela コードによりすべての初期状況とすべての可能な実行が検証され，図 6 の検証アルゴリズムの実現となる．なお，Promela 言語とこのコードの解説は紙面の都合上省略する．

検証は PC/AT 互換機（OS：FreeBSD 3.2，CPU：Pentium III 450 Mhz，メモリ：1 G バイト）上で，Spin バージョン 3.3.3 および GCC 2.95.2 を用いて

表 1 リーダー選出アルゴリズム[16] の検証に要した時間とメモリ（POR: partial order reduction）
Table 1 Time and memory consumed to verify the leader election algorithm[16]. (POR: partial order reduction)

プロセス数	$(G \wedge L) \vee (\neg G \wedge \neg L)$ の検査				無限ループの検査	
n	POR あり		POR なし			
5	2.3 Mb	0.2 Sec	2.1Mb	0.2 Sec	3.4 Mb	1.0 Sec
6	1.7 Mb	0.1 Sec	1.8 Mb	0.1 Sec	2.0 Mb	0.5 Sec
7	261.7 Mb	168.7 Sec	226.5 Mb	187.2 Sec	592.2 Mb	1125.0 Sec
8	89.7 Mb	18.9 Sec	97.4 Mb	34.71 Sec	116.3 Mb	80.9 Sec
9	312.1 Mb	526.3 Sec	401.0 Mb	756.3 Sec	601.2 Mb	1280.4 Sec
10*	8.0 Mb	11417.4 Sec	8.5 Mb	17268.8 Sec	8.0 Mb	36079.0 Sec

表 2 安定彩色アルゴリズム[24] の検証に要した時間とメモリ（POR: partial order reduction）
Table 2 Time and memory consumed to verify the coloring algorithm[24]. (POR: partial order reduction)

プロセス数	$(G \wedge L) \vee (\neg G \wedge \neg L)$ の検査				無限ループの検査	
n	POR あり		POR なし			
5	3.7 Mb	0.6 Sec	3.3 Mb	0.7 Sec	6.7 Mb	3.0 Sec
6	35.3 Mb	15.2 Sec	30.0 Mb	17.7 Sec	79.5 Mb	65.4 Sec
7*	0.7 Mb	1960.1 Sec	0.6 Mb	2026.0 Sec	0.7 Mb	6566.8 Sec
8*	3.9 Mb	43955.8 Sec	3.9 Mb	45754.6 Sec	3.9 Mb	150298.3 Sec
9*	4.7 Mb	81213.1 Sec	4.7 Mb	72232.6 Sec	1.6 Mb	341695.4 Sec

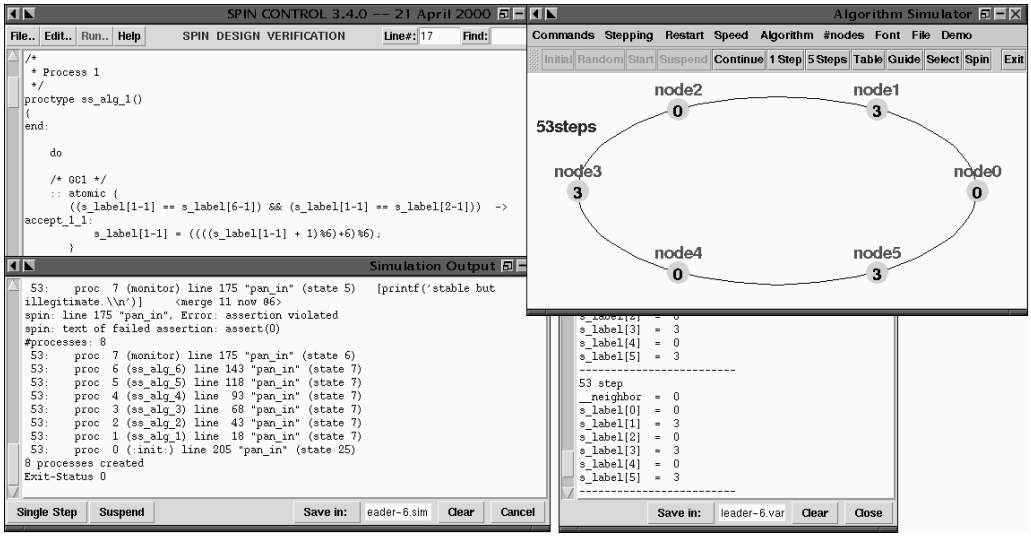

図 8 対話的検証環境の例
Fig. 8 An interactive verification environment.

行った．

表 1 に文献 16) のリーダー選出アルゴリズムの検証結果を示す．このアルゴリズムはプロセス数 n が素数であると仮定している（n が合成数のときは，アルゴリズムが存在しないことが知られている）．n が素数の場合，アルゴリズムが正しいことを確認できた．n が合成数の場合での検証は，デッドロックとなる実行例を出力して停止した．**表 2** には，文献 24) の自己安定彩色アルゴリズムの検証結果を示す．

いずれの n に対しても，アルゴリズムが正しいことが検証できた．なお表 1，表 2 ともに，n に*を付けた場合の検証は Spin の状態圧縮機能を使用したことを示す．使用メモリは大きく削減できる反面，実行時間が増大している．なお $\neg G(\gamma) \wedge \neg L(\gamma)$ の検証には partial order reduction 機能を用いた場合と用いなかった場合両方で検証し，その実行時間と消費メモリを測定した．

図 8 に，我々が文献 29) で報告した自己安定分散

アルゴリズムの可視化システムを拡張し，本検証システムと連動させた対話的環境を示す．この画面例では文献 16) のリーダー選出アルゴリズムの検証結果を，アルゴリズムが存在しないことが知られている $n = 6$ の場合に対して実行した様子を示している．本環境により，アルゴリズムがデッドロックになる実行過程を対話的に知ることができる．右上のウインドウはネットワーク（リング）を図示しており，小さな丸がプロセス，円弧がリンクを，そして小さな丸の中の数字はプロセスの局所状態を示している．右下のウインドウは，デッドロックに至った実行の過程を示している．各実行ステップごとでの各プロセスの状態を，ウインドウをスクロールすることで知ることができる（左上のウインドウは Promela 言語に変換された後の自己安定分散アルゴリズムの記述を示し，その下のウインドウが Spin の出力である）．

6. 結　　論

本論文では，自己安定分散システムの検証と可視化を行うシステムを提案した．我々のシステムは検証のために時間とメモリを消費はするものの，自動的に検証が可能であるという利点がある．提案した言語 Spr は，さまざまなネットワーク形状やプロセス数に容易に対応が可能な言語であり，分散アルゴリズム理論の研究では特に有用である．また可視化部により，検証結果の確認が容易となった．

文献 27) では，Sun Microsystems 社 SPARCStation 20 上で行われたモデル検証系 SMV による検証では，$n = 7$ の場合に 40363.1 秒要し，$n = 8$ の場合では 10 時間以内に検証が終了しなかったと報告されている．一方，我々のシステムでは $n = 7$ の場合に $168.7 + 1125.0 = 1293.7$ 秒要し，$n = 9$ の場合では 1806.7 秒で検証を完了している．CPU 性能比[23] を 10 倍と多めに見積もっても，Spin を用いた我々の方が早く検証を終えている．しかし実行はモデル検証系の違いおよび搭載メモリ量の違いなどに強く依存するので，本論文の手法と文献 27) での手法を上記の結果をもって単純比較するべきではないと考える．

また文献 27) において本論文と同じように Spin を用い，文献 7) の相互排除アルゴリズムの検証も行っている．$n \leq 6$ に対して partial order reduction を用いた場合と用いない場合で検証を行い，実行時間を基に partial order reduction の効果はないと述べている（特に $n = 6$ の場合は状態圧縮が行われている）．

一方，我々は文献 16) の相互排除アルゴリズムを検証したところ（表 1），ある程度の partial order reduction の効果が見られることが分かる．表 2 も同様である．しかし表 2 での $n = 9$ のときだけは partial order reduction の効果なく，逆に実行時間が大きくなっている．これらのことから，自己安定分散アルゴリズムが対象のとき，Spin は以下のような特性を持っていると結論できる．n が小さいときは，partial order reduction の効果はそのオーバヘッドと同程度と思われる．しかし n が増えてくると，少しずつ効果が現れる．さらに n が増えた場合，使用メモリを主記憶容量内におさえるために状態圧縮をして実験を行うと，partial order reduction を行うときの状態圧縮/伸長のオーバヘッドにより，その効果が消えてしまう．

今後の課題として，使用メモリを削減する手法の考案があげられる．本検証システムは，実行スケジューラとして C デーモンのみを想定している．D デーモンの場合は探索すべき状態空間が，C デーモンの場合よりもさらに増大するからである．今後は D デーモンを想定した検証法の検討も必要である．また本システムは状態通信モデルを採用しているが，レジスタ通信モデルやメッセージ伝達モデルを想定した検証システムの検討も今後の課題である．

謝辞　本研究の一部は，文部省科学研究費補助金（奨励研究 A，課題番号 11780229）の支援を受けている．

参 考 文 献

1) Arora, A. and Gouda, M.G.: Distributed Reset, *IEEE Trans. Comput.*, Vol.43, pp.1026–1038 (1994).
2) Bal, H.E., Steiner, J.G. and Tanenbaum, A.S.: Programming Languages for Distributed Computing Systems, *ACM Computing Surveys*, Vol.21, No.3, pp.261–322 (1989).
3) Beauquier, J., Berard, B. and Fribourg, L.: A New Rewrite Method for Proving Convergence of Self-Stabilizing Systems, *13th International Symposium on Distributed Computing* (*DISC*), LNCS 1693, pp.240–253, Springer-Verlag (1999).
4) Beauquier, J., Gradinariu, M. and Johnen, C.: Randomized Self-Stabilizing Optimal Leader Election under Arbitrary Scheduler on Rings, Technical Report 1225, LRI (1999).
5) Chow, R. and Johnson, T.: *Distributed Operating Systems & Algorithms*, Addison Wesley (1997).
6) Clarke, Jr., E.M., Grumberg, O. and Peled, D.A.: *Model Checking*, The MIT Press (1999).
7) Dijkstra, E.W.: Self-Stabilizing Systems in Spite of Distributed Control, *Comm. ACM*,

Vol.17, No.11, pp.643–644 (1974).
8) Dolev, S., Israeli, A. and Moran, S.: Self Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity, *Proc. 9th ACM Symposium on Principles of Distributed Computing*, pp.103–117, ACM (1990).
9) Gouda, M.G.: The Triumph and Tribulation of System Stabilization, *Proc. International Workshop on Distributed Algorithms* (*WDAG 95*), LNCS 972, pp.1–18 (1995).
10) Holzmann, G.J.: *Design and Validation of Computer Protocols*, Prentice Hall (1991).
11) Holzmann, G.J.: The Model Checker Spin, *IEEE Trans. Softw. Eng.*, Vol.23, No.5, pp.279–295 (1997).
12) Holzmann, G.J. and Peled, D.: An Improvement in Formal Verification, *FORTE* (1994).
13) Holzmann, G.J. and Puri, A.: A Minimized Automaton Representation of Reachable States, *International Journal on Software Tools for Technology Transfer*, Vol.2, No.3, pp.270–278 (1999).
14) Hsu, S.-C. and Huang, S.-T.: Analyzing Self-Stabilization with Finite-State Machine Model, *Proc. International Conference of Distributed Computing Systems*, pp.624–631 (1992).
15) Hsu, S.-C. and Huang, S.-T.: A Self-Stabilizing Algorithm for Maximal Matching, *Information Processing Letters*, Vol.43, pp.77–81 (1992).
16) Huang, S.: Leader Election in Uniform Rings, *ACM Trans. Prog. Lang. Syst.*, Vol.15, No.3, pp.563–573 (1993).
17) Huth, M.R.A. and Ryan, M.D.: *Logic in Computer Science*, Cambridge University Press (2000).
18) Kessels, J.L.W.: An Exercise in Proving Self-Stabilization with a Variant Function, *Information Processing Letters*, Vol.29, No.1, pp.39–42 (1988).
19) Lakhnech, Y. and Siegel, M.: Deductive Verification of Stabilizing Systems, *Proc. 2nd Workshop on Self-Stabilizing Systems* (*WSS97*), pp.201–216 (1997).
20) Lynch, N.A.: *Distributed Algorithms*, Morgan Kaufmann (1996).
21) Schneider, M.: Self-Stabilization, *ACM Computing Surveys*, Vol.25, No.1, pp.45–67 (1993).
22) Shukla, S.K., Rosenkrantz, D.J. and Ravi, S.S.: Simulation and Validation Tool for Self-Stabilizing Protocols, *Spin Workshop* (1996).
23) Standard Performance Evaluation Corporation: http://www.spec.org/.
24) Sur, S. and Srimani, P.K.: A Self-Stabilizing Algorithm for Coloring Bipartite Graphs, *Information Sciences*, Vol.69, pp.219–227 (1993).
25) Tel, G.: *Introduction to Distributed Algorithms*, 2nd edition, Cambridge University Press (2000).
26) The Web Page of Spin: http://netlib.bell-labs.com/netlib/spin/whatispin.html.
27) Tsuchiya, T., Nagano, S., Paidi, R.B. and Kikuno, T.: Symbolic Model Checking for Self-Stabilizing Algorithms, *IEEE Trans. Parallel and Distributed Systems*, Vol.12, No.1, pp.81–95 (2001).
28) 山下悟史：自己安定アルゴリズムの可視化システムの開発，広島大学工学部卒業論文 (2000).
29) 山下悟史，角川裕次，阿江 忠：自己安定分散アルゴリズムの可視化システムの開発，情報処理学会全国大会，pp.2Q–04 (2000).
30) 角川裕次：自己安定分散アルゴリズムの自動検証システム，ソフトウエア工学研究会，Vol.2000-SE-128, pp.9–16 (2000).

(平成 13 年 3 月 8 日受付)
(平成 14 年 3 月 14 日採録)

角川 裕次（正会員）

1990 年山口大学工学部電子工学科卒業．1992 年広島大学大学院工学研究科情報工学専攻博士課程前期修了．同大学助手，講師を経て，現在大学院工学研究科情報工学専攻助教授．分散アルゴリズム，教育工学，多言語情報処理の研究を行う．博士（工学）．

山下 悟史

1977 年生．平成 12 年広島大学工学部第二類（電気系）情報工学課程卒業．現在三菱電機インフォメーションシステムズ（株）勤務．