課題研究報告書

分散リーダエレクションプロトコルの形式仕様とモデル検査

小椋 友芳

主指導教員 緒方 和博 教授

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和６年３月

Abstract

In this research project, we created formal specifications of some leader election algorithms and then verified that they enjoy desired properties using model checking. By doing this, we will show how to create a formal specification of a leader election algorithm and how to model check that the algorithm enjoys desired properties based on the formal specification in a realistic execution time.

In distributed systems, centralized management is not a good idea in terms of system load and failure, and it is considered better to design a system with multiple processes that have the same role. However, it is easier to make sure to keep data integrity if there is one special process that is in charge of it.

The process of selecting one elected process, or "leader," from among multiple processes that comprise a distributed system is called leader election. Various algorithms have been proposed for leader election.

Since leader election has been used in recent years for distributed databases and for building fault tolerance into systems, the accuracy of the leader election algorithm has become very important from the perspective of system operation. For this reason, leader election algorithms are modeled and model checking tools are used to verify that such algorithms satisfy desired properties. However, the difficulty of modeling and the unrealistically long verification time due to state explosion have become problems.

In the experiments described in this report, we created formal specifications using Maude for three leader election algorithms: the Bully algorithm, the Chang-Roberts algorithm, and the Franklin algorithm. In addition, based on the created formal specifications, we used model checking to verify whether each algorithm satisfies the guaranteed properties.

In the Bully algorithm, when a process detects that a leader has stopped, it starts the next election process. The process that started the election sends a message to all processes whose IDs are greater than its own ID. If there is no reply from the processes to which the message has been sent, the process that started the election becomes the leader. Conversely, if a process whose ID is greater than the process that initiated the current election replies to the message, the election is replaced. Eventually, there will be no processes left that reply to such a message, and the one remaining process will become the leader. In this report, we formalize and specify the Bully algorithm, which performs the above operations, as a state transition system with eight observable components, and 11 rewriting rules for state transitions. We then used model checking to verify that the algorithm enjoys desired properties based on the formal specification created. In the verification experiments, the number of processes targeted by the Bully algorithm was set to five, and model checking was performed. The results of the

validation experiments confirm that the Bully algorithm satisfies the properties of a leader election algorithm.

In the Bully algorithm, all processes perform the selection process step by step in synchronized timing. Therefore, it is difficult to perform model checking in Maude, which inherently performs the process asynchronously. However, in this report, we decided to perform model checking of the Bully algorithm by performing three synchronous measures to the rewriting rules of the formal specification that we created. As a result of the measures, it was confirmed that the properties of a leader election algorithm were satisfied, so the measures are considered to be effective.

In the Chang-Roberts algorithm, when a non-candidate process detects the absence of a leader, it starts the election process. The process that detects the absence of the leader becomes a candidate as a starting process and sends a message including its own process ID to the neighboring process. If the process that received the message is a non-candidate, the message is relayed to the neighboring process, and if the process that received the message is a candidate, it compares its own process ID with the process ID in the received message. If its own process ID is higher, it sends the received message to its neighbor. Conversely, if its own process ID is lower, it discards the received message. This process is continued until only the message containing the lowest process ID has gone around the ring. In this report, we formalize and specify the Chang-Roberts algorithm, which operates as described above, as a state transition system with four observable components, and nine rewriting rules for state transitions. The five properties that the algorithm must satisfy are described as LTL formulas. We then used model checking to verify that the algorithm enjoys the five properties based on the formal specification created. In the verification experiments, the number of processes targeted by the Chang-Roberts algorithm was set to five, two initial states were defined that considered the sorting order, and model checking was performed. The results of the verification experiments confirmed that the Chang-Roberts algorithm satisfies the properties of a leader election algorithm, regardless of the order of processes arranged in a ring. Additionally, it was confirmed that the three properties specific to the Chang-Roberts algorithm were also satisfied.

In the Franklin algorithm, a non-initiator process starts the election process when it detects the absence of a leader. The process that detects the absence of the leader becomes the initiator and sends a message including its own process ID to both neighboring processes. If a process receiving the message is a non-initiator, it relays the message to the neighboring process as a passive process. However, if a process that received the message is an initiator, it compares its own process ID with the higher of the process IDs of the messages received from

both neighboring processes. If its own process ID is higher, it sends a message including its own process ID to both neighboring processes again. If its own process ID is lower, it becomes a passive process. If it is equal to its own process ID, it becomes a leader. In this report, we formalize and specify the Franklin algorithm, which operates as described above, as a state transition system with four observable components, and 12 rewriting rules for state transitions. The two properties that the algorithm must satisfy are described as LTL formulas. We then used model checking to verify that the algorithm enjoys the two properties based on the formal specification created. In the verification experiments, the number of processes targeted by the Franklin algorithm was set to five, two initial states were defined that considered the sorting order, and model checking was performed. The results of the verification experiments confirmed that the Franklin algorithm satisfies the properties of a leader election algorithm, regardless of the order of processes arranged in a ring.

keywords: model checking, distributed algorithm, leader election, Maude, safety property, liveness property, state transition system

概要

本研究課題では、リーダ選出アルゴリズムについて形式仕様を作成し、アルゴリズムの正当性の性質をモデル検査で検証することで、現実的な時間で検証可能なリーダ選出アルゴリズムの形式仕様の作成方法とモデル検査の実施方法を示す。

分散システムでは、システムへの負荷や故障の観点から、集中管理は好ましくなく、同じ役割を持つ複数のプロセスでシステムを設計する方がよいとされてる。しかし、データの整合性担保などは選任のプロセスがいた方が、処理が容易になる。

分散システムを構成する複数のプロセスの中から、1つの選任プロセス、つまり「リーダ」を選ぶ処理をリーダ選出という。リーダ選出のアルゴリズムとしては、これまでに、様々なアルゴリズムが提案されている。

近年は分散データベースやシステムの耐障害性の組み込みなどでリーダ選出が利用されているため、リーダ選出アルゴリズムの正確性はシステム運用の視点からも非常に重要になっている。そのため、リーダ選出アルゴリズムをモデル化し、モデル検査ツールを利用して、そのようなアルゴリズムが所望の性質を満たしていることを検証されることが増えてきた。しかしながら、モデル化の難しさや、状態爆発による非現実的な検証時間の長さが問題になっている。

本稿の実験ではブリーアルゴリズム、Chang-Roberts アルゴリズム、Franklin アルゴリズムの 3 つのリーダ選出アルゴリズムについて Maude にて形式仕様を作成した。また、作成した形式仕様を基に、リーダ選出アルゴリズムの正当性の性質や各アルゴリズムが、保証すべき性質が満たされているかの確認をモデル検査で検証した。

ブリーアルゴリズムでは、あるプロセスがリーダの停止を検出すると次の選任処理を開始する。選任を開始したプロセスは、自身の ID より大きな ID をもつすべてのプロセスにメッセージを送信する。メッセージを送ったプロセスから返信がない場合は、選任のプロセスがリーダになる。反対に、選任のプロセスより大きな ID をもつプロセスから返信があった場合は、選任を交代する。最終的には、返信するプロセスがなくなり、残った 1 つのプロセスがリーダになる。本稿では、上述の動作をするブリーアルゴリズムを、8 つの観測可能成分で状態遷移システムとして形式化し、11 の書き換え規則で形式仕様として表現した。また、作成した形式仕様を用いて満たすべき性質の確認をモデル検査で検証した。検証の実験では、ブリーアルゴリズムの対象となるプロセス数を 5 つにして、モデル検査を行った。検証実験の結果、ブリーアルゴリズムは、リーダ選出アルゴリズムとしての性質が満たされていることを確認できた。

ブリーアルゴリズムでは、すべてのプロセスが、同期的にタイミングを合わせて 1 ステップずつ選任処理を進める。そのため、本来、非同期で処理を実施する Maude ではモデル検査を行うことが難しい。だが、本稿では作成した形式仕様の書き換え規則に、3 つの同期的な対応を行うことで、ブリーアルゴリズムのモデル検査を行うことにした。対応の結果、リーダ選出アルゴリズムとしての性質を満

たしていることが確認できたため、対応には効果があったと考える。

Chang-Roberts アルゴリズムは、非候補者のプロセスが、リーダの不在を検知すると選任処理を開始する。リーダの不在を検知したプロセスは、始動プロセスとして候補者となり、隣のプロセスに自身のプロセス ID を含むメッセージを送る。メッセージを受け取ったプロセスが非候補者の場合は、メッセージをそのまま隣のプロセスに中継し、メッセージを受け取ったプロセスが候補者の場合は、自身のプロセス ID と受信したメッセージのプロセス ID を比較する。自身のプロセス ID の方が大きい場合は、受け取ったメッセージを隣に送る。反対に、自身のプロセス ID の方が小さい場合は、受け取ったメッセージを破棄する。この処理を最小のプロセス ID を含むメッセージだけがリングを一周するまで行う。本稿では、上述の動作をする Chang-Roberts アルゴリズムを、4 つの観測可能成分で状態遷移システムとして形式化し、9 の書き換え規則で形式仕様として表現した。アルゴリズムが満たすべき 5 つの性質は、LTL 式で記述している。また、作成した形式仕様を用いて満たすべき性質の確認をモデル検査で検証した。検証の実験では、Chang-Roberts アルゴリズムの対象となるプロセス数を 5 つにし、並び順を考慮した初期状態を 2 つ定義して、モデル検査を行った。検証実験の結果、Chang-Roberts アルゴリズムは、リング状に配置されたプロセスの並び順に関係なく、リーダ選出アルゴリズムとしての正当性の性質が満たされていることを確認できた。また、Chang-Roberts アルゴリズム固有の 3 つの性質についても満たされていることが確認できた。

Franklin アルゴリズムは、非イニシエーターのプロセスが、リーダの不在を検知すると選任処理を開始する。リーダの不在を検知したプロセスは、イニシエーターとなり、両隣のプロセスに自身のプロセス ID を含むメッセージを送る。メッセージを受け取ったプロセスが非イニシエーターの場合は、パッシブプロセスとしてメッセージをそのまま隣のプロセスに中継するが、メッセージを受け取ったプロセスがイニシエーターの場合は、両隣のプロセスから受信したメッセージのプロセス ID のうち、大きな方のプロセス ID と自身のプロセス ID を比較する。自身のプロセス ID の方が大きい場合は、再度、両隣のプロセスに自身のプロセス ID を含むメッセージを送る。自身のプロセス ID の方が小さい場合は、パッシブプロセスとなる。自身のプロセス ID と等しい場合はリーダとなる。本稿では、上述の動作をする Franklin アルゴリズムを、4 つの観測可能成分で状態遷移システムとして形式化し、12 の書き換え規則で形式仕様として表現した。アルゴリズムが満たすべき 2 つの性質は、LTL 式で記述している。また、作成した形式仕様を用いて満たすべき性質の確認をモデル検査で検証した。検証の実験では、Franklin アルゴリズムの対象となるプロセス数を 5 つにし、並び順を考慮した初期状態を 2 つ定義して、モデル検査を行った。検証実験の結果、Franklin アルゴリズムは、リング状に配置されたプロセスの並び順に関係なく、リーダ選出アルゴリズムとしての正当性の性質が満たされていることを確認できた。

キーワード： モデル検査、分散アルゴリズム、リーダ選出、Maude、安全性、

活性、状態遷移システム

目 次

図 目 次

表 目 次

第1章　はじめに

1.1　背景

分散システムでは、システムへの負荷や故障の観点から、集中管理は好ましくなく、同じ役割を持つ複数のプロセスでシステムを設計する方がよいとされてる。しかし、場合によっては選任のプロセスがいた方が、処理が容易になることがある。例えば、システムにおける共有資源の割り当てやデータの整合性担保などは、選任のプロセスがいた方が、処理が容易になる。

分散システムを構成する複数のプロセスの中から、1つの選任プロセス、つまり「リーダ」を選ぶ処理をリーダ選出という。リーダ選出のアルゴリズムとしては、これまでに、無向リングや完全グラフ、グリッドなど、様々なネットワークトポロジでのアルゴリズムが提案されている。また、近年では、分散合意プロトコルである Paxos[7, 8] のコンセンサスと安全性の保証を利用したリーダ選出や、同じく分散合意プロトコルである Raft[9] の Leader Election 機能によるリーダ選出が、分散データベースなどで利用されている。

現在では、前述の分散データベースの利用や、システムの耐障害性の組み込みなどでリーダ選出が利用されているため、そのアルゴリズムの正確性はシステムの視点からも非常に重要になっている。そのため、リーダ選出アルゴリズムの正確性をモデル化し、TLA+などのツールを利用して検証されている [12]。しかしながら、モデル化の難しさ [13] や、状態爆発による非現実的な検証時間の長さ [14] が問題になっている。

1.2　目的

本研究課題は、幾つかの異なるリーダ選出アルゴリズムにおいて形式仕様を作成し、その正当性の性質をモデル検査で検証することで、現実的な時間で検証できるリーダ選出アルゴリズムの形式仕様の作成とモデル検査の方法を示すことを目的とする。

本稿では、これまでに提案された3つのリーダ選出アルゴリズム、ブリーアルゴリズム [1]、Chang-Roberts アルゴリズム [4] および Franklin アルゴリズム [6] の形式仕様を Maude[10] を使って作成する。また、作成した形式仕様に基づいて、リーダ選出アルゴリズムとしての正当性の性質や、アルゴリズム固有の性質が満たさ

れているかどうかの確認を、モデル検査を用いて検証する。リーダ選出アルゴリズムとしての正当性の性質とは、「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質と「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質のことである。

本稿の実験で利用した環境は以下の通りである。

- MacBook Pro (16-inch, 2021)
  - 10 コア CPU、24 コア GPU、16 コア Neural Engine 搭載 Apple M1 Max
  - 64GB ユニファイドメモリ
- モデル検査器
  - Maude 3.2.1

1.3 構成

本稿の構成は以下の通りである。

- 第 1 章 - 本研究課題の背景や目的などを説明する。
- 第 2 章 - 本稿で必要になる知識や用語について説明する。
- 第 3 章 - ブリーアルゴリズムの形式仕様と実施したモデル検査の結果について説明する。
- 第 4 章 - Chang-Roberts アルゴリズムの形式仕様と実施したモデル検査の結果について説明する。
- 第 5 章 - Franklin アルゴリズムの形式仕様と実施したモデル検査の結果について説明する。
- 第 6 章 - 本研究課題についてのまとめと課題について報告する。

第2章　予備知識

本章では、本稿で行う Maude によるリーダ選出アルゴリズムの形式仕様の作成や、満たすべき性質をモデル検査で検証するために必要な知識や用語の説明を行う。

2.1　状態機械

状態機械は、いくつかの状態と状態間の遷移で構成される。状態機械について定義を行う。S は状態の集合、I は初期状態の集合で $I \subseteq S$ であり、T は状態間の二項関係で $T \subseteq S \times S$ である。この時、状態機械 M は、$M \triangleq \langle S, I, T \rangle$ である。T の各要素 $(s, s') \in T$ は、s から s' への状態遷移とよび、$s \to s'$ で表す。

2.2　クリプキ構造

クリプキ構造は、システムの振る舞いを表現して、そのモデルを検証する際に使用する。クリプキ構造について定義を行う。S は状態の集合、I は初期状態の集合で $I \subseteq S$ であり、P は原子命題の集合である。L は型が $S \to 2^P$ のラベリング関数、T は状態間の左全体二項関係で $T \subseteq S \times S$ である。この時、クリプキ構造 K は、$K \triangleq \langle S, I, P, L, T \rangle$ である。なお、クリプキ構造は、LTL モデル検査のモデルである。

2.3　ラベル付きクリプキ構造

ラベル付きクリプキ構造 [16] を用いることで、公平性の仮定の記述が可能になる。ラベル付きクリプキ構造について定義を行う。ラベル付きクリプキ構造は、クリプキ構造にイベント（状態遷移の名前）の集合を加えて定義する。lS は状態の集合、lI は初期状態の集合で、$lI \subseteq lS$ である。lE はイベントの集合、lP は原子命題の集合で、$lP \cap lE = \emptyset$ である。lL は型が $lS \to 2^{lP}$ のラベリング関数、lT は状態間の全域的な三項関係で $lT \subseteq lS \times lE \times lS$ である。この時、ラベル付きクリプキ構造 lK は、$lK \triangleq \langle lS, lI, lE, lP, lL, lT \rangle$ である。lT の各要素 $(s, e, s') \in lT$ は、s から s' へのラベル付き遷移 e とよび、$s \to_e s'$ で表す。また、$s, s' \in lS$ に対して、$(s, \iota, s') \notin lT$ となるようなイベント $\iota \in lE$ が存在する。

2.4 EES クリプキ構造

ラベル付きクリプキ構造は、クリプキ構造の状態にイベントを埋め込むことにより模倣が可能である。これは「ラベル付きクリプキ構造の events-embedded-in-states クリプキ構造」（EES クリプキ構造）[16] と呼ばれる。EES クリプキ構造について定義を行う。$S_{ees} = \{(e, s) \mid e \in lE, s \in lS\}$ は状態の集合、$I_{ees} = \{(\iota, s) \mid s \in lI\}$ は初期状態の集合、$P_{ees} = lP \cup lE$ は原子命題の集合である。$L_{ees}((e, s))$ は、各 $e \in lE$ に対して、$\{e\} \cup lL(s)$ となるラベリング関数、$T_{ees} = \{((e, s), (e', s')) \mid e, e' \in lE, s, s' \in lS, (s, e', s') \in lT\}$ は、状態間の全域的な二項関係の集合である。この時、EES クリプキ構造 K_{ees} は、$K_{ees} \triangleq \langle S_{ees}, I_{ees}, P_{ees}, L_{ees}, T_{ees} \rangle$ である。

2.5 公平性

公平性は、ある条件下において、特定の状態が何度も発生する性質を示すことであり、弱公平性と強公平性がある。弱公平性は「ある条件が有効であり続けるとすれば、最終的に特定の状態が発生する」性質を示すことであり、強公平性は「ある条件が繰り返し有効になるとすれば、最終的に特定の状態が発生する」性質を示すことである。

ラベル付き遷移における公平性の仮定の定義を行う。ラベル付き遷移 e が、実行可能状態であることを $enabled(e)$、直前に実行されたことを $applied(e)$ で表すとする。この時、弱公平性は、$\Diamond\Box enabled(e) \Rightarrow \Box\Diamond applied(e)$ であり、強公平性は、$\Box\Diamond enabled(e) \Rightarrow \Box\Diamond applied(e)$ である。

2.6 Maude

Maude は、書き換え論理に基づく仕様およびプログラミング言語であり、処理系も同じく Maude という。モデル検査を行う機能として、`search` コマンドと LTL モデル検査器を標準機能として提供する。[11]

2.7 スープと観測可能成分

本稿における状態の表現について説明する。状態をデータ構造として表現する方法は数多くあるが、本稿では、結合法則と交換方法を満たす、名前と値のペアのコレクションを波括弧（ブレース）で囲んだもので状態を表現する。また、用語として、結合法則と交換方法を満たすコレクションをスープ（soup）、名前と値のペアを観測可能成分（observable component）とする。したがって、状態は観測可能成分のスープをブレースで囲ったものとして表現される。ソースコード 2.1 は、本稿における状態表現の例である。`proc[0]: normal` が観測可能

成分、(proc[0]: normal) (proc[1]: normal) (proc[2]: leader) がスープ、{(proc[0]: normal) (proc[1]: normal) (proc[2]: leader)}が状態をそれぞれ表している。

ソースコード 2.1: 本稿における状態の表現

```
{(proc[0]: normal) (proc[1]: normal) (proc[2]: leader)}
```

2.8 書き換え規則

状態遷移を表す書き換え規則は、rl キーワードを利用して、$rl[L] : LHS => RHS.$ の形式で記述する。L はラベルを表している。この書き換え規則は、$=>$ 記号の左辺の LHS 項を右辺の RHS 項に書き換えることを示している。また、条件付きの書き換えは、crl キーワードを利用して、$crl[L] : LHS => RHS\ if\ C.$ の形式で記述する。この書き換え規則は、if キーワードで指定した条件 C が成り立つ時に、$=>$ 記号の左辺の LHS 項を右辺の RHS 項に書き換えることを示している。if キーワードの条件は複数指定することが可能である。複数の条件を指定する場合は、$crl[L] : LHS => RHS\ if\ C_1/\backslash ... /\backslash C_n.$ の形式で記述する。この場合、条件が成り立つのは、$C_1, ..., C_n$ のすべてが成り立つ場合である。

2.9 search コマンド

Maude の search コマンドは、与えた状態から状態遷移により到達可能な状態を幅優先で探索し、ある条件を満たす状態があるかどうかを調べる。なお、探索の深さの上界を指定することも可能である。

ソースコード 2.2: search コマンドの例

```
search [1] in EXSAMPLE : init =>*
 {(proc[0]: normal) (proc[1]: normal) (proc[2]: leader)} .
```

ソースコード 2.2 は、search コマンドの例である。EXSAMPLE は架空のモジュールで、init は検索を開始する状態である。このコマンドにより、init 状態から到達可能な範囲に{(proc[0]: normal) (proc[1]: normal) (proc[2]: leader)}の状態があるかどうかを調べることができる。search コマンドの後の [1] は、該当する状態を 1 つ見つけた場合、処理を停止させる指示である。

2.10 LTL モデル検査器

Maude が提供している LTL モデル検査器を利用するには、Maude に付随している model-checker.maude ファイルを in コマンドを使って読み込む必要がある。

model-checker.maude ファイルには、状態と原子命題の充足関係を表す|=演算子が定義されている SATISFACTION モジュールや、LTL 式を簡略化する等式が定義されている LTL-SIMPLIFIER モジュール、LTL モデル検査で利用する modelCheck 関数が定義されている MODEL-CHECKER モジュールが記述されている。

第3章　ブリーアルゴリズムの形式仕様とモデル検査

本章では、リーダ選出アルゴリズムの最初の事例としてブリーアルゴリズムを取り上げる。最初にブリーアルゴリズムの概要について説明する。次に Maude で記述した形式仕様について説明した後、最後に作成した形式仕様を基に実施したモデル検査と結果について説明する。

3.1　ブリーアルゴリズムの概要

ブリーアルゴリズム [2, 3] は、Garcia-Molina によって考案されたリーダ選出のアルゴリズムである。このアルゴリズムでは、前提として各プロセスは比較可能な一意の ID を持っている。また、各プロセスは他のそれぞれのプロセスの ID を知っている。しかし、各プロセスはどのプロセスが動作中で、どのプロセスが停止中であるかは分からない。

あるプロセスがリーダの停止を検出すると次の選任処理を開始する。

1. 選任を開始したプロセスは、自身の ID より大きな ID をもつすべてのプロセスに Election メッセージを送信する。
2. Election メッセージを送ったどのプロセスからも OK メッセージの返信がない場合は、選任のプロセスがリーダになる。
3. 選任のプロセスより大きな ID をもつプロセスから OK メッセージの返信があった場合は、選任を交代する。

プロセスは自身の ID より小さい ID をもつプロセスから Election メッセージを受信した場合、OK メッセージを返信し、選任を引き継ぐことになる。最終的に OK メッセージを返信するプロセスがなくなり、残った 1 つのプロセスがリーダになる。リーダになったプロセスは他のすべてのプロセスに Coordinator メッセージを送信してリーダになったことを知らせる。

図 3.1 に、ブリーアルゴリズムがどのように動作するかを示す。すべてのプロセスは同期的に処理をすすめ、直接、他のプロセスにメッセージを送っている。図 3.1(a) では、プロセス 4 がリーダの停止を検出し、自身の ID より大きな ID をも

つ、プロセス 5、6、7 に Election メッセージを送信する。Election メッセージを受け取ったプロセス 5 と 6 は、選任を引き継ぐため、OK メッセージをプロセス 4 に返信する。図 3.1(b) では、選任を引き継いだプロセス 5 が、プロセス 6 と 7 に、同じく選任を引き継いだプロセス 6 がプロセス 7 に Election メッセージを送信する。プロセス 6 は選任を引き継ぐため、OK メッセージをプロセス 5 に返信する。また、プロセス 6 は自身より大きな ID を持つプロセス 7 から OK メッセージの返信がないことを確認し、自身がリーダになることを知る。図 3.1(c) では、プロセス 6 が自身がリーダになったことを他のすべてのプロセスに Coordinator メッセージを送信して知らせる。

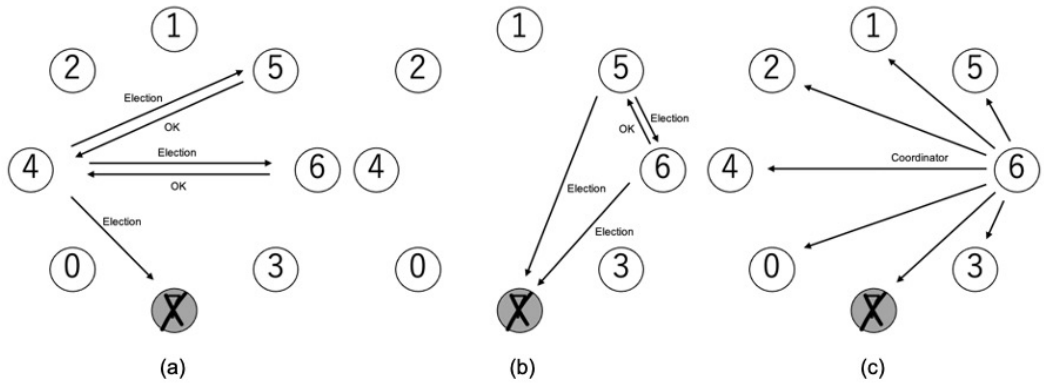

図 3.1: ブリーアルゴリズム

3.2 ブリーアルゴリズムの形式仕様

3.2.1 観測可能成分

ブリーアルゴリズムでは、ソースコード 3.1 に示す 8 つの観測可能成分を用いて状態遷移システムを形式化した。

ソースコード 3.1: ブリーアルゴリズムの観測可能成分

```
1 op proc[_]:_ : Nat ProcState -> OComp [ctor] .
2 op curLeader[_]:_ : Nat Nat -> OComp [ctor] .
3 op sndElectionMsgCnt[_]:_ : Nat Nat -> OComp [ctor] .
4 op rcvOkMsgCnt[_]:_ : Nat Nat -> OComp [ctor] .
5 op rcvTimeoutMsgCnt[_]:_ : Nat Nat -> OComp [ctor] .
6 op procIds:_ : Set{Nat} -> OComp [ctor] .
7 op network:_ : Soup{Message} -> OComp [ctor] .
8 op tran:_ : LabeledEvent -> OComp [ctor] .
```

`ProcState` はプロセスの状態、`OComp` は観測可能成分、`Message` はメッセージをそれぞれ表すソートである。

proc[_]:_ は、第1引数にプロセスID、第2引数にプロセスの状態をとる。プロセスの状態はleader、normal、initiator、failedLeaderのいずれかになる。この観測可能成分は、プロセスの状態を表しており、第1引数のプロセスIDを持つプロセスが、第2引数の状態であることを意味する。

curLeader[_]:_ は、第1引数および第2引数にプロセスIDをとる。この観測可能成分は、現在、リーダと認識しているプロセスを表しており、第1引数のプロセスIDを持つプロセスが、第2引数のプロセスIDを持つプロセスを現在、リーダと認識していることを意味する。

sndElectionMsgCnt[_]:_ は、第1引数にプロセスID、第2引数にプロセスが送信したElectionメッセージ数をとる。この観測可能成分は、選任で送ったElectionメッセージ数を表しており、第1引数のプロセスIDを持つプロセスが、選任で他のプロセスに送ったElectionメッセージ数は、第2引数の値であることを意味する。

rcvOkMsgCnt[_]:_ は、第1引数にプロセスID、第2引数にプロセスが受信したOKメッセージ数をとる。この観測可能成分は、選任で受け取ったOKメッセージ数を表しており、第1引数のプロセスIDを持つプロセスが、選任で他のプロセスから受け取ったOKメッセージ数は、第2引数の値であることを意味する。

rcvTimeoutMsgCnt[_]:_ は、第1引数プロセスID、第2引数にプロセスが受信したTimeoutメッセージ数をとる。この観測可能成分は、選任で受け取ったTimeoutメッセージの数を表しており、第1引数のプロセスIDを持つプロセスが、選任で他のプロセスから受け取ったTimeoutメッセージ数は、第2引数の値であることを意味する。

procIds:_ は、引数にプロセスIDの集合をとる。この観測可能成分は、ブリーアルゴリズムの対象になるプロセス一覧を表し、引数の値が対象のプロセスのIDであることを意味する。

network:_ は、引数にメッセージのスープをとる。この観測可能成分は、各プロセスが選任処理で送受信するメッセージのためのネットワークを表し、引数のメッセージがネットワーク内にあるを意味する。

tran:_ は、引数に書き換え規則に対応するラベル付きイベントをとる。この観測可能成分は、直前に実行された書き換え規則が、ラベル付きイベントに対応する書き換え規則であることを意味する。

ブリーアルゴリズムでは、各プロセスが任意のプロセスに対してメッセージを送信するが、本稿の形式仕様ではネットワークトポロジのモデル化は行わない。メッセージに送信元と送信先のプロセスIDを設定することで、ネットワークノード間のリンクを表現し、各プロセスが処理対象となるメッセージを特定可能にしている。

3.2.2 書き換え規則

ブリーアルゴリズムを形式仕様として表現するために、以降に示す 11 の書き換え規則を利用する。また、これらの書き換え規則では表 3.1 の変数を用いている。

表 3.1: ブリーアルゴリズムの書き換え規則で用いる変数

変数名	説明
S、S0、S1	観測可能成分の集合の変数
ID、LID、ID0、ID1	プロセス ID の変数
SEC	送信した Election メッセージ数の変数
ROC	受信した OK メッセージ数の変数
RTC	受信した Timeout メッセージ数の変数
IDS	プロセス ID の集合の変数
NW	メッセージの集合の変数
E	ラベル付きイベントの変数

ブリーアルゴリズムの書き換え規則では、直前に実行された書き換え規則を示す観測可能成分に、自身の書き換え規則に対応したラベル付きイベントを設定する。ソースコード 3.2 は、ラベル付きイベントの定義である。`notran` は、直前に実行された書き換え規則がないことを表すラベル付きイベントで、定数として定義している。また、それ以外のラベル付きイベントは、プロセス ID をとり、ラベル付きイベントを返す演算として定義している。本稿の書き換え規則のラベルは、複数の単語をハイフンでつなげた形式としている。演算として定義しているラベル付きイベントの名前は、対応した書き換え規則のラベルの各単語の頭文字をつなげたものにしている。例えば、書き換え規則 `become-failed-leader` に対応したラベル付きイベントの名前は、`bfl` となる。

ソースコード 3.2: ラベル付きイベントの定義

```
1 sort LabeledEvent .
2 op notran : -> LabeledEvent .
3 ops bfl bi se nee nie et iee
4  ieo iet ibn ibl : Nat -> LabeledEvent .
```

ソースコード 3.3: become-failed-leader の定義

```
1 rl [become-failed-leader] :
2  {(proc[ID]: leader) (network: empty) (tran: E) S} =>
3  {(proc[ID]: failedLeader) (network: empty)
4   (tran: bfl(ID)) S} .
```

ソースコード 3.3 は、書き換え規則 `become-failed-leader` の定義である。この書き換え規則は、`leader` 状態のプロセス ID が、ネットワークにメッセージがない（`empty`）場合に故障し、`failedLeader` 状態になることを表している。

ブリーアルゴリズムでは、すべてのプロセスは同期的に、つまり、すべてのプロセスがタイミングを合わせて 1 ステップずつ処理をすすめる。あるステップでプロセスがメッセージを送信した場合、その次のステップには送信先のプロセスからメッセージが返信される。それゆえ、プロセスが任意のタイミングでメッセージを処理することはない。しかし、Maude によるモデル検査では非同期に処理を実行するため、プロセスが任意のタイミングでメッセージを処理することになる。そのため、以前に実施された選任で送信したメッセージが、ネットワークに残った状態で leader 状態のプロセスが故障し、次の選任が開始される場合がある。この場合、ネットワークに残ったメッセージにより、選任が正しく行われない。したがって、ネットワークが空である場合に、leader 状態のプロセスが故障することとしている。

ソースコード 3.4: become-initiator の定義

```
1 rl [become-initiator] :
2  {(proc[ID]: normal) (proc[LID]: failedLeader)
3   (curLeader[ID]: LID) (sndElectionMsgCnt[ID]: SEC)
4   (rcvOkMsgCnt[ID]: ROC) (rcvTimeoutMsgCnt[ID]: RTC)
5   (tran: E) S} =>
6  {(proc[ID]: initiator) (proc[LID]: failedLeader)
7   (curLeader[ID]: LID) (sndElectionMsgCnt[ID]: 0)
8   (rcvOkMsgCnt[ID]: 0) (rcvTimeoutMsgCnt[ID]: 0)
9   (tran: bi(ID)) S} .
```

ソースコード 3.4 は、書き換え規則 become-initiator の定義である。この書き換え規則は、normal 状態のプロセス ID が、自身がリーダと認識しているプロセス LID の故障に気づいた場合に、initiator 状態になることを表している。また、プロセスの状態変更に合わせ、プロセス ID が送受信した各メッセージ数を初期化するため、(sndElectionMsgCnt[ID]: 0)、(rcvOkMsgCnt[ID]: 0) および (rcvTimeoutMsgCnt[ID]: 0) としている。

ソースコード 3.5: start-election の定義

```
1 crl [start-election] :
2  {(proc[ID]: initiator) (sndElectionMsgCnt[ID]: SEC)
3   (procIds: IDS) (network: NW) (tran: E) S} =>
4  {(proc[ID]: initiator)
5   (sndElectionMsgCnt[ID]: | biggerThanIds(ID, IDS) |)
6   (procIds: IDS) (network: (
7     sndElectionMsgs(ID, biggerThanIds(ID, IDS)) NW
8   )) (tran: se(ID)) S}
9  if SEC == 0 .
```

ソースコード 3.5 は、書き換え規則 start-election の定義である。この書き換え規則は、initiator 状態のプロセス ID が、まだ選任を開始していない、つまり Election メッセージを送信していない場合に選任を開始し、自身のプロセス ID よ

り大きなIDを持つプロセスに対してElectionメッセージを送信することを表している。sndElectionMsgsは、指定した複数のプロセスに対してElectionメッセージを送信する。また、biggerThanIdsは、ブリーアルゴリズムの対象のプロセスの中で、自身のプロセスIDより大きなプロセスIDの集合である。sndElectionMsgsをソースコード3.6、biggerThanIdsをソースコード3.7のように定義する。

ソースコード 3.6: sndElectionMsgsの定義

```
op sndElectionMsgs : Nat Set{Nat} -> Soup{Message} .
eq sndElectionMsgs(FID, empty) = empty .
eq sndElectionMsgs(FID, (TID, IDS)) = message(FID, TID,
   election) sndElectionMsgs(FID, IDS) .
```

ソースコード 3.7: biggerThanIdsの定義

```
op makeBiggerThan : Nat Nat Set{Nat} -> Set{Nat} .
op makeBiggerThanIds : Nat Set{Nat} Set{Nat} -> Set{Nat} .
op biggerThanIds : Nat Set{Nat} -> Set{Nat} .

eq makeBiggerThan(N0, N1, S) =
 if N0 < N1 then insert(N1, S) else S fi .
eq makeBiggerThanIds(N0, (N, S), S0) =
 if S == empty then makeBiggerThan(N0, N, S0)
 else makeBiggerThanIds(N0, S, makeBiggerThan(N0, N, S0))
 fi .
eq biggerThanIds(N, S) = makeBiggerThanIds(N, S, empty) .
```

ソースコード 3.8: normal-execution-electionの定義

```
crl [normal-execution-election] :
 {(proc[ID0]: normal) (proc[LID]: failedLeader)
  (curLeader[ID0]: LID) (sndElectionMsgCnt[ID0]: SEC)
  (rcvOkMsgCnt[ID0]: ROC) (rcvTimeoutMsgCnt[ID0]: RTC)
  (network: (message(ID1, ID0, election) NW))
  (tran: E) S} =>
 {(proc[ID0]: initiator) (proc[LID]: failedLeader)
  (curLeader[ID0]: LID) (sndElectionMsgCnt[ID0]: 0)
  (rcvOkMsgCnt[ID0]: 0) (rcvTimeoutMsgCnt[ID0]: 0)
  (network: (sndOkMsg(ID0, ID1) NW)) (tran: nee(ID0)) S}
 if ID0 > ID1 .
```

ソースコード3.8は、書き換え規則normal-execution-electionの定義である。この書き換え規則は、normal状態のプロセスID0が、自身のプロセスIDより小さなプロセスIDを持つプロセスID1から送信されたElectionメッセージmessage(ID1, ID0, election)を受信した場合、選任を引き継ぎ、initiator状態になることを表している。また、プロセスID0は、Electionメッセージの送信元ID1へ選任を引き継ぐため、OKメッセージsndOkMsg(ID0, ID1)を送信する。プロセスの状態変更に合わせ、プロセスID0が送受信した各メッセージ数を

初期化するため、(sndElectionMsgCnt[ID]: 0)、(rcvOkMsgCnt[ID]: 0) および (rcvTimeoutMsgCnt[ID]: 0) としている。

ソースコード 3.9: normal-ignore-election の定義

```
crl [normal-ignore-election] :
 {(proc[ID0]: normal) (curLeader[ID0]: LID)
  (proc[LID]: leader)
  (network: (message(ID1, ID0, election) NW))
  (tran: E) S} =>
 {(proc[ID0]: normal) (curLeader[ID0]: LID)
  (proc[LID]: leader) (network: NW) (tran: nie(ID0)) S}
 if ID0 > ID1 .
```

ソースコード 3.9 は、書き換え規則 normal-ignore-election の定義である。この書き換え規則は、選任によりプロセス LID が leader 状態となった後に、normal 状態のプロセス ID0 が、自身のプロセス ID より小さなプロセス ID を持つプロセス ID1 から送信された Election メッセージ message(ID1, ID0, election) を受信したことを表している。すでに選任が終了しているため、この Election メッセージに意味はない。したがって、この Election メッセージは破棄し、ネットワークから削除する。ブリーアルゴリズムでは選任でリーダが決定した後に、プロセスが Election メッセージを受け取ることはない。しかし、Maude によるモデル検査は非同期に処理を実行するため、任意のタイミングで Election メッセージを受け取ることになる。この書き換え規則は、Maude でブリーアルゴリズムのモデル検査を行うための規則である。

ソースコード 3.10: election-timeout の定義

```
crl [election-timeout] :
 {(proc[ID0]: failedLeader)
  (network: (message(ID1, ID0, election) NW))
  (tran: E) S} =>
 {(proc[ID0]: failedLeader)
  (network: (sndTimeoutMsg(ID0, ID1) NW))
  (tran: et(ID0)) S}
 if ID0 > ID1 .
```

ソースコード 3.10 は、書き換え規則 election-timeout の定義である。この書き換え規則は、failedLeader 状態のプロセス ID0 が、自身のプロセス ID より小さなプロセス ID を持つプロセス ID1 から送信された Election メッセージ message (ID1, ID0, election) を受信した場合、プロセス ID1 に Timeout メッセージ sndTimeoutMsg(ID0, ID1) を送信することを表している。ブリーアルゴリズムでは、任意のプロセスに Election メッセージを送信した次のステップに OK メッセージが返信されない場合、タイムアウトと考える。本稿の形式仕様では、Timeout メッセージを返信することでタイムアウトを表現する。

ソースコード 3.11: initiator-execution-election の定義

```
1 crl [initiator-execution-election] :
2  {(proc[ID0]: initiator)
3   (network: (message(ID1, ID0, election) NW))
4   (tran: E) S} =>
5  {(proc[ID0]: initiator)
6   (network: (sndOkMsg(ID0, ID1) NW)) (tran: iee(ID0)) S}
7  if ID0 > ID1 .
```

ソースコード 3.11 は、書き換え規則 initiator-execution-election の定義である。この書き換え規則は、initiator 状態のプロセス ID0 が、自身のプロセス ID より小さなプロセス ID を持つプロセス ID1 から送信された Election メッセージ message(ID1, ID0, election) を受信した場合、選任を引き継ぐため、Election メッセージの送信元 ID1 へ OK メッセージ sndOkMsg(ID0, ID1) を送信することを表している。プロセス ID0 は、既に自身によりリーダプロセスの故障に気づいたか、他のプロセスからの Election メッセージによって、initiator 状態になっているため、プロセスの状態変更や送受信した各メッセージ数の初期化は行わない。

ソースコード 3.12: initiator-execution-ok の定義

```
1 crl [initiator-execution-ok] :
2  {(proc[ID0]: initiator) (rcvOkMsgCnt[ID0]: ROC)
3   (network: (message(ID1, ID0, ok) NW)) (tran: E) S} =>
4  {(proc[ID0]: initiator) (rcvOkMsgCnt[ID0]: ROC + 1)
5   (network: NW) (tran: ieo(ID0)) S}
6  if ID0 < ID1 .
```

ソースコード 3.12 は、書き換え規則 initiator-execution-ok の定義である。この書き換え規則は、initiator 状態のプロセス ID0 が、自身のプロセス ID より大きなプロセス ID を持つプロセス ID1 から送信された OK メッセージ message(ID1, ID0, ok) を受信した場合、受信した OK メッセージ数 ROC をインクリメントすることを表している。

ソースコード 3.13: initiator-execution-timeout の定義

```
1 crl [initiator-execution-timeout] :
2  {(proc[ID0]: initiator) (rcvTimeoutMsgCnt[ID0]: RTC)
3   (network: (message(ID1, ID0, timeout) NW))
4   (tran: E) S} =>
5  {(proc[ID0]: initiator) (rcvTimeoutMsgCnt[ID0]: RTC + 1)
6   (network: NW) (tran: iet(ID0)) S}
7  if ID0 < ID1 .
```

ソースコード 3.13 は、書き換え規則 initiator-execution-timeout の定義である。この書き換え規則は、initiator 状態のプロセス ID0 が、自身のプロセス ID より大きなプロセス ID を持つプロセス ID1 から送信された Timeout メッセージ message(ID1, ID0, timeout) を受信した場合、受信した Timeout メッセージ数 RTC をインクリメントすることを表している。

ソースコード 3.14: initiator-become-normal の定義

```
crl [initiator-become-normal] :
 {(proc[ID0]: initiator) (sndElectionMsgCnt[ID0]: SEC)
  (rcvOkMsgCnt[ID0]: ROC) (rcvTimeoutMsgCnt[ID0]: RTC)
  (tran: E) S} =>
 {(proc[ID0]: normal) (sndElectionMsgCnt[ID0]: SEC)
  (rcvOkMsgCnt[ID0]: ROC) (rcvTimeoutMsgCnt[ID0]: RTC)
  (tran: ibn(ID0)) S}
 if SEC > 0 and SEC == (ROC + RTC) and ROC > 0 .
```

ソースコード 3.14 は、書き換え規則 initiator-become-normal の定義である。この書き換え規則は、initiator 状態のプロセス ID0 が、選任で送信した Election メッセージ数 SEC と受信した OK メッセージ数 ROC および Timeout メッセージ数 RTC の合計が等しく、OK メッセージを受信している、つまり、自身のプロセス ID より大きなプロセス ID を持つ他のプロセスに選任が引き継がれている場合に、プロセス ID0 は normal 状態になることを表している。

ブリーアルゴリズムでは選任で他のプロセスに Election メッセージを送信した場合、次のステップには OK メッセージを受信するか、応答なし（タイムアウト）として判定する。それゆえに、選任が他のプロセスに引き継がれているかは、次のステップで判断することができる。この書き換え規則では、選任で送信した Election メッセージに対する返信メッセージがすべて揃い、OK メッセージを受信したことを確認するまで、プロセスが normal 状態にならないように、同期的に行う必要がある。本稿のブリーアルゴリズムの形式仕様では、メッセージの送受信を同期的に行うために、各プロセスが選任で送信した Election メッセージ数、受信した OK メッセージ数および Timeout メッセージ数を状態に含め、各メッセージを送受信する書き換え規則でメッセージ数を変更している。この書き換え規則では、状態に含まれる任意のプロセスが選任で送受信した各メッセージ数を利用して、送信した Election メッセージに対する返信メッセージがすべて揃い、OK メッセージを受信したことを確認する条件を記述している。

ソースコード 3.15: initiator-become-leader の定義

```
crl [initiator-become-leader] :
 {(proc[ID0]: initiator) (curLeader[ID0]: ID1)
  (sndElectionMsgCnt[ID0]: SEC) (rcvTimeoutMsgCnt[ID0]: RTC)
  (rcvOkMsgCnt[ID0]: ROC) (procIds: IDS) (tran: E) S} =>
 {(proc[ID0]: leader) (curLeader[ID0]: ID0)
  (sndElectionMsgCnt[ID0]: SEC) (rcvTimeoutMsgCnt[ID0]: RTC)
  (rcvOkMsgCnt[ID0]: ROC) (procIds: IDS) (tran: ibl(ID0))
  syncCurLeader(S, ID0, smallerThanIds(ID0, IDS))}
 if SEC > 0 and SEC == RTC and ROC == 0 .
```

ソースコード 3.15 は、書き換え規則 initiator-become-leader の定義である。この書き換え規則は、initiator 状態のプロセス ID0 が、選任で送信した Election メッセージ数 SEC と受信した Timeout メッセージ数 RTC が等しく、OK メッセー

ジを受信していない、つまり、選任を交代する他のプロセスがいない場合に、プロセス ID0 は leader 状態になることを表している。また、プロセスの状態変更にあわせ、プロセス ID0 が現在、リーダと認識しているプロセスに自身を設定するとともに、syncCurLeader(S, ID0, smallerThanIds(ID0, IDS)) を用いて、自身のプロセス ID より小さな ID を持つプロセスが、現在、リーダと認識しているプロセスにも自身を設定している。

ブリーアルゴリズムにおいて、任意のプロセスがリーダになるのは、選任で送信した Election メッセージに対して、OK メッセージを返信するプロセスがなくなり、最後に残った 1 つのプロセスになった場合である。そのため、この書き換え規則は、選任で送信した Election メッセージに対する返信メッセージがすべて揃い、OK メッセージを受信していないことを確認するまで、プロセスが leader 状態にならないように、同期的に行う必要がある。本稿のブリーアルゴリズムの形式仕様では、メッセージの送受信を同期的に行うために、各プロセスが選任で送信した Election メッセージ数、受信した OK メッセージ数および Timeout メッセージ数を状態に含め、各メッセージを送受信する書き換え規則でメッセージ数を変更している。この書き換え規則では、状態に含まれる任意のプロセスが選任で送受信した各メッセージ数を利用して、送信した Election メッセージに対する返信メッセージがすべて揃い、OK メッセージを受信していないことを確認する条件を記述している。

ブリーアルゴリズムでは、任意のプロセスがリーダになった場合、他のすべてのプロセスに Coordinator メッセージを送信する。リーダ以外のプロセスは、Coordinator メッセージを受信し、新たなリーダを知る。この処理は同じステップで同期的に行われる。しかし Maude では非同期に処理を実行するため、Coordinator メッセージの受信は任意のタイミングになり、すべてのプロセスが同期的に新たなリーダを知ることは難しい。そのため、本稿では新たにリーダになったプロセスより小さなプロセス ID を持つプロセスが、リーダと認識しているプロセスを syncCurLeader を用いて書き換えることで、Coordinator メッセージを送受信し、新たなリーダを知る処理の代わりとしている。

syncCurLeader をソースコード 3.16、smallerThanIds をソースコード 3.17 のように定義する。syncCurLeader は、指定した複数のプロセスに対して、現在、リーダと認識しているプロセスに、指定したプロセスを設定する。また、smallerThanIds は、ブリーアルゴリズムの対象のプロセスの中で、指定したプロセス ID より小さなプロセス ID の集合である。syncCurLeader の第 1 引数には curLeader を含むの観測可能成分のスープを、第 2 引数にはリーダプロセスのプロセス ID を、第 3 引数には書き換え対象となるプロセス ID の集合をそれぞれ渡す。

ソースコード 3.16: syncCurLeader の定義

```
1 op syncCurLeader : Soup{OComp} Nat Set{Nat} -> Soup{OComp} .
2
3 eq syncCurLeader(empty, ID, IDS) = empty .
```

```
 4 eq syncCurLeader(S, ID, empty) = S .
 5 eq syncCurLeader((curLeader[ID0]: ID1) S, ID, IDS) =
 6  if ID0 in IDS then
 7   (curLeader[ID0]: ID) syncCurLeader(S, ID, IDS)
 8  else (curLeader[ID0]: ID1)
 9   syncCurLeader(S, ID, IDS)
10  fi .
11 eq syncCurLeader((proc[ID0]: initiator) S, ID, IDS) =
12  (proc[ID0]: normal) syncCurLeader(S, ID, IDS) .
13 eq syncCurLeader(S0 S1, ID, IDS) =
14  S0 syncCurLeader(S1, ID, IDS) [owise] .
```

ソースコード 3.17: smallerThanIds の定義

```
 1 op makeSmallerThan : Nat Nat Set{Nat} -> Set{Nat} .
 2 op makeSmallerThanIds : Nat Set{Nat} Set{Nat} -> Set{Nat} .
 3 op smallerThanIds : Nat Set{Nat} -> Set{Nat} .
 4
 5 eq makeSmallerThan(N0, N1, S) =
 6  if N0 > N1 then insert(N1, S) else S fi .
 7 eq makeSmallerThanIds(N0, (N, S), S0) =
 8   if S == empty then makeSmallerThan(N0, N, S0)
 9   else makeSmallerThanIds(N0, S, makeSmallerThan(N0, N, S0))
10   fi .
11 eq smallerThanIds(N, S) = makeSmallerThanIds(N, S, empty) .
```

3.2.3 LTL 式

ブリーアルゴリズムが、活性の性質を満たすことをモデル検査するために、以降に示す LTL 式を利用する。

ソースコード 3.18: BULLY-PREDS モジュールの定義

```
 1 mod BULLY-PREDS is
 2  pr BULLY .
 3  inc SATISFACTION .
 4  subsort BState < State .
 5  vars ID : Nat .
 6  var E : LabeledEvent .
 7  var S : Soup{OComp} .
 8  var PROP : Prop .
 9  op leader : Nat -> Prop .
10  op enabled : LabeledEvent -> Prop .
11  op applied : LabeledEvent -> Prop .
12  eq {(proc[ID]: leader) S} |= leader(ID) = true .
13  eq {(tran: bi(ID)) S} |= enabled(ibl(ID)) = true .
14  eq {(tran: nee(ID)) S} |= enabled(ibl(ID)) = true .
```

```
15  eq {(tran: E) S} |= applied(E) = true .
16  eq {S} |= PROP = false [owise] .
17 endm
```

ソースコード 3.18 は、BULLY-PREDS モジュールの定義である。このモジュールは、ブリーアルゴリズムの性質を LTL 式で表現するために必要な原子命題を定義している。

BULLY-PREDS は、BULLY および SATISFACTION をインポートしている。BULLY は、3.2.2 項で説明した書き換え規則を定義したモジュールである。SATISFACTION は、Maude で LTL モデル検査器を利用するために必要な model-checker.maude に定義されているモジュールで、状態と原子命題の充足関係を表す|=演算子が記述されている。BULLY-PREDS では、leader、enabled および applied の 3 つの演算と |=演算子を利用して、LTL 式に必要な状態と原子命題の充足関係を定義している。例えば、状態が{(proc[ID]: leader) S}の場合に、原子命題 leader(ID) は充足する。なお、leader はプロセスがリーダであることを、enabled はラベル付きイベントが実行可能状態あることを、applied はラベル付きイベントが直前に実行されたことをそれぞれ表す演算である。また、ラベル付イベント ibl、つまり書き換え規則 initiator-become-leader が実行されるには、書き換え規則 become-initiator または、normal-execution-election が実行され、プロセスの状態が initiator となっている必要がある。そのため、ラベル付イベント bi および nee をラベル付イベント ibl が実行可能状態とする充足関係に含めている。

ソースコード 3.19: BULLY-CHECK モジュールの定義

```
 1 mod BULLY-CHECK is
 2  inc BULLY-PREDS .
 3  inc MODEL-CHECKER .
 4  inc LTL-SIMPLIFIER .
 5  var E : LabeledEvent .
 6  op leaderLiveness : -> Formula .
 7  op sf : LabeledEvent -> Formula .
 8  op sfair : -> Formula .
 9  eq leaderLiveness = <> (leader(0) \/ leader(1) \/
10   leader(2) \/ leader(3) \/ leader(4)) .
11  eq sf(E) = ([] <> enabled(E)) -> ([] <> applied(E)) .
12  eq sfair = sf(ibl(0)) /\ sf(ibl(1)) /\ sf(ibl(2)) /\
13   sf(ibl(3)) /\ sf(ibl(4)) .
14 endm
```

ソースコード 3.19 は、BULLY-CHECK モジュールの定義である。このモジュールは、モデル検査用のモジュールでブリーアルゴリズムの性質を表す LTL 式を定義している。

BULLY-CHECK は、BULLY-PREDS、MODEL-CHECKER および LTL-SIMPLIFIER をインポートしている。MODEL-CHECKER および LTL-SIMPLIFIER は、model-checker.maude に定義されているモジュールである。MODEL-CHECKER には、LTL モデル検査

で利用する modelCheck 関数が記述されている。LTL-SIMPLIFIER には、LTL モデル検査のための補助モジュールで、LTL 式を簡略化する等式が定義されている。

BULLY-CHECK では、LTL 式を 3 つを定義している。1 つ目の leaderLiveness は、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質を表す LTL 式である。2 つ目の sf は、強公平性を意味する LTL 式である。3 つ目の sfair は、ラベル付イベント ibl に対して強公平性を仮定する LTL 式である。本稿の形式仕様を用いて活性のモデル検査を行った場合、書き換え規則による遷移がループしてしまい、書き換え規則 initiator-become-leader が実行されずに反例が検出されることになる。この問題に対応するために、ラベル付きイベント ibl に対して、強公平性を仮定した。プロセスの状態が leader に遷移するのは、書き換え規則 initiator-become-leader が実行された場合である。そのため、ラベル付イベント ibl に対して強公平性を仮定している。

3.3 ブリーアルゴリズムのモデル検査

本節では、前節で説明した形式仕様に基づいて、ブリーアルゴリズムが、リーダ選出アルゴリズムとしての正当性を満たしているかの確認をモデル検査で検証する。

リーダ選出アルゴリズムの正当性では、次の 2 つの性質を検証する。

- 「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質
- 「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質

モデル検査を実施するにあたり、2 つ初期状態を定義する。1 つ目は、「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質のモデル検査で利用する初期状態 init である。2 つ目は、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質のモデル検査で利用する初期状態 init2 である。ブリーアルゴリズムの対象になるプロセスが 5 つで、それぞれのプロセス ID が 0 から 4 の場合、初期状態 init および初期状態 init2 を下記のように定義する。

ソースコード 3.20: 初期状態 init の定義

```
1 op {_} : Soup{OComp} -> BState [ctor] .
2 op init : -> BState .
3 eq init = {
4  (proc[0]: normal) (proc[1]: normal) (proc[2]: normal)
5  (proc[3]: normal) (proc[4]: leader)
6  (curLeader[0]: 4) (curLeader[1]: 4) (curLeader[2]: 4)
7  (curLeader[3]: 4) (curLeader[4]: 4)
```

```
 (sndElectionMsgCnt[0]: 0) (sndElectionMsgCnt[1]: 0)
 (sndElectionMsgCnt[2]: 0) (sndElectionMsgCnt[3]: 0)
 (sndElectionMsgCnt[4]: 0)
 (rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0)
 (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]: 0)
 (rcvOkMsgCnt[4]: 0)
 (rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 0)
 (rcvTimeoutMsgCnt[2]: 0) (rcvTimeoutMsgCnt[3]: 0)
 (rcvTimeoutMsgCnt[4]: 0)
 (procIds: (0, 1, 2, 3, 4))
 (network: empty)
 (tran: notran)
} .
```

ソースコード 3.21: 初期状態 init2 の定義

```
op init2 : -> BState .
eq init2 = {
 (proc[0]: normal) (proc[1]: normal) (proc[2]: normal)
 (proc[3]: normal) (proc[4]: failedLeader)
 (curLeader[0]: 4) (curLeader[1]: 4) (curLeader[2]: 4)
 (curLeader[3]: 4) (curLeader[4]: 4)
 (sndElectionMsgCnt[0]: 0) (sndElectionMsgCnt[1]: 0)
 (sndElectionMsgCnt[2]: 0) (sndElectionMsgCnt[3]: 0)
 (sndElectionMsgCnt[4]: 0)
 (rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0)
 (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]: 0)
 (rcvOkMsgCnt[4]: 0)
 (rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 0)
 (rcvTimeoutMsgCnt[2]: 0) (rcvTimeoutMsgCnt[3]: 0)
 (rcvTimeoutMsgCnt[4]: 0)
 (procIds: (0, 1, 2, 3, 4))
 (network: empty)
 (tran: notran)
} .
```

ソースコード 3.20 は初期状態 `init`、ソースコード 3.21 は初期状態 `init2` の定義である。2 つの初期状態の差異は、プロセスの状態である。初期状態 `init` では、プロセスの状態は、プロセス ID が 4 のプロセスを `leader` 状態とし、その他のプロセスについては `normal` 状態とする。初期状態 `init2` では、プロセスの状態は、プロセス ID が 4 のプロセスを `failedLeader` 状態とし、その他のプロセスについては `normal` 状態とする。初期状態 `init2` において、プロセス ID が 4 のプロセスを `failedLeader` 状態としたのは、初期状態で「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質を満たすことを避けるためである。現在、リーダと認識しているプロセスについては、すべてのプロセスで 4 をリーダと認識している。選任で送信した Election メッセージ数、受信した OK メッセー

ジ数および Timeout メッセージ数については 0 とする。ブリーアルゴリズムの対象になるプロセス一覧は、0 から 4 までの ID を持つ 5 つのプロセスである。ネットワークワークは empty とし、ネットワークにメッセージはないものとする。また、直前に実行された書き換え規則はなしとする。

ブリーアルゴリズムのリーダ選出アルゴリズムとしての正当性を検証する。「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質は、ソースコード 3.22 のコマンドを、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質は、ソースコード 3.23 のコマンドをそれぞれ実行することで確認できる。

ソースコード 3.22: 正当性における安全性の性質を確認するコマンド

```
1 search [1] in BULLY : init =>*
2   {(proc[ID0]: leader) (proc[ID1]: leader) S} .
```

ソースコード 3.23: 正当性における活性の性質を確認するコマンド

```
1 red in BULLY-CHECK : modelCheck(init2, sfair ->
     leaderLiveness) .
```

ソースコード 3.22 およびソースコード 3.23 のコマンドを実行して、Maude から得られる結果は以下の通りである。

```
search [1] in BULLY : init =>* {S (proc[ID0]: leader) proc[
   ID1]: leader} .

No solution.
states: 846912  rewrites: 68014564 in 2986938ms cpu (3002461
   ms real) (22770 rewrites/second)
==========================================
reduce in BULLY-CHECK : modelCheck(init2, sfair ->
   leaderLiveness) .
rewrites: 68667974 in 3733396ms cpu (3745548ms real) (18392
   rewrites/second)
result Bool: true
```

ソースコード 3.22 のコマンドの実行結果は、初期状態 init から到達可能な範囲に、2 つの異なるプロセスが同時にリーダになる状態への遷移は見つからなかったことを示している。これにより、「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質が満たされることが分かった。次に、ソースコード 3.23 のコマンドの実行結果は、反例は検出されず、モデル検査が成功している。これにより、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質が満たされることが分かった。これらの結果から、ブリーアルゴリズムのリーダ選出アルゴリズムとしての正当性の性質が、満たされていることが確認できた。

3.4 まとめ

本章では、ブリーアルゴリズムの形式仕様を作成し、モデル検査を行った。モデル検査の実験では、ブリーアルゴリズムの対象となるプロセスを5つとし、リーダ選出アルゴリズムとしての正当性が満たされることを検証した。リーダ選出アルゴリズムとしての正当性は、「常に2つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質と「いつか少なくとも1つのプロセスが、リーダに選ばれる」という活性の性質を検証し、2つの性質が満たされることを確認した。これにより、ブリーアルゴリズムは、リーダ選出アルゴリズムとしての正当性を満たしていることを確認できた。

ブリーアルゴリズムは、すべてのプロセスが、同期的にタイミングを合わせて1ステップずつ処理を行うため、本来、非同期で処理を実施するMaudeではモデル検査を行うことが難しい。だが、作成した形式仕様の書き換え規則に同期的に遷移を進める対応を行うことで、ブリーアルゴリズムのモデル検査を行うことにした。本稿の形式仕様の書き換え規則で行った遷移を同期的に進める対応は次の3つである。

1つ目の対応は、書き換え規則`initiator-become-normal`で行っている。この書き換え規則は、選任で自身のプロセスIDより大きなIDをもつプロセスにElectionメッセージを送信した後、何れかのプロセスがOKメッセージを返したため、選任は他のプロセスに引き継がれたとして、自身は`normal`状態になる書き換え規則である。この書き換え規則を同期的に行い、選任で送信したElectionメッセージに対する返信メッセージがすべて揃い、OKメッセージを受信したことを確認するまで、プロセスが`normal`状態にならないようにしている。

2つ目の対応は、書き換え規則`initiator-become-leader`で行っている。この書き換え規則は、選任で自身のプロセスIDより大きなIDをもつプロセスにElectionメッセージを送信した後、何れのプロセスもOKメッセージを返さなかったため、自身が`leader`状態になる書き換え規則である。この書き換え規則を同期的に行い、選任で送信したElectionメッセージに対する返信メッセージがすべて揃い、OKメッセージを受信していないことを確認するまで、プロセスが`leader`状態にならないようにしている。本稿の形式仕様では、これら2つの書き換え規則を同期的に行えるように、各プロセスが選任で送信したElectionメッセージ数、受信したOKメッセージ数およびTimeoutメッセージ数を状態に持っており、各メッセージを送受信する書き換え規則でメッセージ数の更新を行っている。これら2つの書き換え規則では、状態に含まれる、任意のプロセスが選任で送受信した各メッセージ数を利用して、書き換え規則の条件を記述し、同期的に処理を行うようにしている。

3つ目の対応は、2つ目と同じく書き換え規則`initiator-become-leader`で行っている。ブリーアルゴリズムでは、プロセスが新たにリーダになる場合、Coordinatorメッセージを各プロセスに送信して、新たなリーダの通知を同期的に行う。

本稿の形式仕様では、各プロセスがリーダと認識しているプロセスIDを状態として持っている。書き換え規則initiator-become-leaderでは、Coordinatorメッセージを各プロセスに送信し、新たなリーダを通知して、各プロセスがリーダと認識しているプロセスIDを変更する代わりに、1つの等式で同期的に変更している。

これらの対応の結果、リーダ選出アルゴリズムとしての正当性の性質を満たしていることが確認できたため、対応には効果があったと考える。

「いつか少なくとも1つのプロセスが、リーダに選ばれる」という活性の性質のモデル検査では、強公平性を仮定して検証を行った。これは強公平性を仮定せずにモデル検査を行った場合、書き換え規則による状態遷移がループしてしまい、反例が検出されるからである。例えば公平性を仮定せずにモデル検査を行った場合は、下記の反例が検出される。なお、下記の結果は、ブリーアルゴリズムの対象となるプロセスを5つにして実施した場合、状態数が多くなり反例が長くなるため、プロセスを4つにして実施している。

```
reduce in BULLY-CHECK : modelCheck(init3, leaderLiveness) .
rewrites: 704 in 2ms cpu (2ms real) (251069 rewrites/second)
result ModelCheckResult: counterexample({{procIds: (0, 1, 2,
    3) network: empty tran: notran (proc[0]: normal) (proc
   [1]: normal) (proc[2]: normal) (proc[3]: failedLeader) (
   curLeader[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (
   curLeader[3]: 3) (sndElectionMsgCnt[0]: 0) (
   sndElectionMsgCnt[1]: 0) (sndElectionMsgCnt[2]: 0) (
   sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt
   [1]: 0) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]: 0) (
   rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 0) (
   rcvTimeoutMsgCnt[2]: 0) rcvTimeoutMsgCnt[3]: 0},'become-
   initiator} {{procIds: (0, 1, 2, 3) network: empty tran:
   bi(0) (proc[0]: initiator) (proc[1]: normal) (proc[2]:
   normal) (proc[3]: failedLeader) (curLeader[0]: 3) (
   curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
   sndElectionMsgCnt[0]: 0) (sndElectionMsgCnt[1]: 0) (
   sndElectionMsgCnt[2]: 0) (sndElectionMsgCnt[3]: 0) (
   rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
   0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
   rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
   rcvTimeoutMsgCnt[3]: 0},'become-initiator} {{procIds: (0,
    1, 2, 3) network: empty tran: bi(1) (proc[0]: initiator)
    (proc[1]: initiator) (proc[2]: normal) (proc[3]:
   failedLeader) (curLeader[0]: 3) (curLeader[1]: 3) (
   curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt[0]:
    0) (sndElectionMsgCnt[1]: 0) (sndElectionMsgCnt[2]: 0) (
   sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt
   [1]: 0) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]: 0) (
   rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 0) (
   rcvTimeoutMsgCnt[2]: 0) rcvTimeoutMsgCnt[3]: 0},'become-
```

```
initiator} {{procIds: (0, 1, 2, 3) network: empty tran:
bi(2) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
 initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 0) (sndElectionMsgCnt[1]: 0) (
sndElectionMsgCnt[2]: 0) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'start-election} {{procIds: (0,
1, 2, 3) network: (message(0, 1, election) message(0, 2,
election) message(0, 3, election)) tran: se(0) (proc[0]:
initiator) (proc[1]: initiator) (proc[2]: initiator) (
proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
3) (curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt
[0]: 3) (sndElectionMsgCnt[1]: 0) (sndElectionMsgCnt[2]:
0) (sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 0) (
rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]:
0) (rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 0) (
rcvTimeoutMsgCnt[2]: 0) rcvTimeoutMsgCnt[3]: 0},'start-
election} {{procIds: (0, 1, 2, 3) network: (message(0, 1,
 election) message(0, 2, election) message(0, 3, election
) message(1, 2, election) message(1, 3, election)) tran:
se(1) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
 initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 0) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'start-election} {{procIds: (0,
1, 2, 3) network: (message(0, 1, election) message(0, 2,
election) message(0, 3, election) message(1, 2, election)
 message(1, 3, election) message(2, 3, election)) tran:
se(2) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
 initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'election-timeout} {{procIds: (0,
 1, 2, 3) network: (message(0, 1, election) message(0, 2,
 election) message(1, 2, election) message(1, 3, election
) message(2, 3, election) message(3, 0, timeout)) tran:
```

```
et(3) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
 initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'election-timeout} {{procIds: (0,
 1, 2, 3) network: (message(0, 1, election) message(0, 2,
 election) message(1, 2, election) message(2, 3, election
) message(3, 0, timeout) message(3, 1, timeout)) tran: et
(3) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'election-timeout} {{procIds: (0,
 1, 2, 3) network: (message(0, 1, election) message(0, 2,
 election) message(1, 2, election) message(3, 0, timeout)
 message(3, 1, timeout) message(3, 2, timeout)) tran: et
(3) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-election} {{
procIds: (0, 1, 2, 3) network: (message(0, 2, election)
message(1, 0, ok) message(1, 2, election) message(3, 0,
timeout) message(3, 1, timeout) message(3, 2, timeout))
tran: iee(1) (proc[0]: initiator) (proc[1]: initiator) (
proc[2]: initiator) (proc[3]: failedLeader) (curLeader
[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (curLeader
[3]: 3) (sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]:
2) (sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-election} {{
procIds: (0, 1, 2, 3) network: (message(1, 0, ok) message
(1, 2, election) message(2, 0, ok) message(3, 0, timeout)
```

```
 message(3, 1, timeout) message(3, 2, timeout)) tran: iee
(2) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-election} {{
procIds: (0, 1, 2, 3) network: (message(1, 0, ok) message
(2, 0, ok) message(2, 1, ok) message(3, 0, timeout)
message(3, 1, timeout) message(3, 2, timeout)) tran: iee
(2) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-ok} {{
procIds: (0, 1, 2, 3) network: (message(2, 0, ok) message
(2, 1, ok) message(3, 0, timeout) message(3, 1, timeout)
message(3, 2, timeout)) tran: ieo(0) (proc[0]: initiator)
 (proc[1]: initiator) (proc[2]: initiator) (proc[3]:
failedLeader) (curLeader[0]: 3) (curLeader[1]: 3) (
curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt[0]:
 3) (sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]: 1) (
sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 1) (rcvOkMsgCnt
[1]: 0) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]: 0) (
rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 0) (
rcvTimeoutMsgCnt[2]: 0) rcvTimeoutMsgCnt[3]: 0},'
initiator-execution-ok} {{procIds: (0, 1, 2, 3) network:
(message(2, 1, ok) message(3, 0, timeout) message(3, 1,
timeout) message(3, 2, timeout)) tran: ieo(0) (proc[0]:
initiator) (proc[1]: initiator) (proc[2]: initiator) (
proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
3) (curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt
[0]: 3) (sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]:
1) (sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 2) (
rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]:
0) (rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 0) (
rcvTimeoutMsgCnt[2]: 0) rcvTimeoutMsgCnt[3]: 0},'
initiator-execution-ok} {{procIds: (0, 1, 2, 3) network:
(message(3, 0, timeout) message(3, 1, timeout) message(3,
 2, timeout)) tran: ieo(1) (proc[0]: initiator) (proc[1]:
```

```
 initiator) (proc[2]: initiator) (proc[3]: failedLeader)
(curLeader[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (
curLeader[3]: 3) (sndElectionMsgCnt[0]: 3) (
sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]: 1) (
sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt
[1]: 1) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]: 0) (
rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 0) (
rcvTimeoutMsgCnt[2]: 0) rcvTimeoutMsgCnt[3]: 0},'
initiator-execution-timeout} {{procIds: (0, 1, 2, 3)
network: (message(3, 1, timeout) message(3, 2, timeout))
tran: iet(0) (proc[0]: initiator) (proc[1]: initiator) (
proc[2]: initiator) (proc[3]: failedLeader) (curLeader
[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (curLeader
[3]: 3) (sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]:
2) (sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 1) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-timeout} {{
procIds: (0, 1, 2, 3) network: message(3, 2, timeout)
tran: iet(1) (proc[0]: initiator) (proc[1]: initiator) (
proc[2]: initiator) (proc[3]: failedLeader) (curLeader
[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (curLeader
[3]: 3) (sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]:
2) (sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 1) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 0)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-timeout} {{
procIds: (0, 1, 2, 3) network: empty tran: iet(2) (proc
[0]: initiator) (proc[1]: initiator) (proc[2]: initiator)
 (proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 1) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-become-normal}, {{
procIds: (0, 1, 2, 3) network: empty tran: ibn(0) (proc
[0]: normal) (proc[1]: initiator) (proc[2]: initiator) (
proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
3) (curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt
[0]: 3) (sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]:
1) (sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 2) (
rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]:
0) (rcvTimeoutMsgCnt[0]: 1) (rcvTimeoutMsgCnt[1]: 1) (
```

```
rcvTimeoutMsgCnt[2]: 1) rcvTimeoutMsgCnt[3]: 0}, 'become-
initiator} {{procIds: (0, 1, 2, 3) network: empty tran:
bi(0) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
 initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 0) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'start-election} {{procIds: (0,
1, 2, 3) network: (message(0, 1, election) message(0, 2,
election) message(0, 3, election)) tran: se(0) (proc[0]:
initiator) (proc[1]: initiator) (proc[2]: initiator) (
proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
3) (curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt
[0]: 3) (sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]:
1) (sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 0) (
rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]:
0) (rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 1) (
rcvTimeoutMsgCnt[2]: 1) rcvTimeoutMsgCnt[3]: 0},'election
-timeout} {{procIds: (0, 1, 2, 3) network: (message(0, 1,
 election) message(0, 2, election) message(3, 0, timeout
)) tran: et(3) (proc[0]: initiator) (proc[1]: initiator)
(proc[2]: initiator) (proc[3]: failedLeader) (curLeader
[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (curLeader
[3]: 3) (sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]:
2) (sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-election} {{
procIds: (0, 1, 2, 3) network: (message(0, 2, election)
message(1, 0, ok) message(3, 0, timeout)) tran: iee(1) (
proc[0]: initiator) (proc[1]: initiator) (proc[2]:
initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 0) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-election} {{
procIds: (0, 1, 2, 3) network: (message(1, 0, ok) message
(2, 0, ok) message(3, 0, timeout)) tran: iee(2) (proc[0]:
 initiator) (proc[1]: initiator) (proc[2]: initiator) (
proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
```

```
3) (curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt
[0]: 3) (sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]:
1) (sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 0) (
rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]:
0) (rcvTimeoutMsgCnt[0]: 0) (rcvTimeoutMsgCnt[1]: 1) (
rcvTimeoutMsgCnt[2]: 1) rcvTimeoutMsgCnt[3]: 0},'
initiator-execution-ok} {{procIds: (0, 1, 2, 3) network:
(message(2, 0, ok) message(3, 0, timeout)) tran: ieo(0) (
proc[0]: initiator) (proc[1]: initiator) (proc[2]:
initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 1) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-ok} {{
procIds: (0, 1, 2, 3) network: message(3, 0, timeout)
tran: ieo(0) (proc[0]: initiator) (proc[1]: initiator) (
proc[2]: initiator) (proc[3]: failedLeader) (curLeader
[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (curLeader
[3]: 3) (sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]:
2) (sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 0) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-timeout} {{
procIds: (0, 1, 2, 3) network: empty tran: iet(0) (proc
[0]: initiator) (proc[1]: initiator) (proc[2]: initiator)
 (proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 1) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-become-normal} {{
procIds: (0, 1, 2, 3) network: empty tran: ibn(1) (proc
[0]: initiator) (proc[1]: normal) (proc[2]: initiator) (
proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
3) (curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt
[0]: 3) (sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]:
1) (sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 2) (
rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]:
0) (rcvTimeoutMsgCnt[0]: 1) (rcvTimeoutMsgCnt[1]: 1) (
rcvTimeoutMsgCnt[2]: 1) rcvTimeoutMsgCnt[3]: 0},'become-
initiator} {{procIds: (0, 1, 2, 3) network: empty tran:
```

```
bi(1) (proc[0]: initiator) (proc[1]: initiator) (proc[2]:
 initiator) (proc[3]: failedLeader) (curLeader[0]: 3) (
curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 0) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 1) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'start-election} {{procIds: (0,
1, 2, 3) network: (message(1, 2, election) message(1, 3,
election)) tran: se(1) (proc[0]: initiator) (proc[1]:
initiator) (proc[2]: initiator) (proc[3]: failedLeader) (
curLeader[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (
curLeader[3]: 3) (sndElectionMsgCnt[0]: 3) (
sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]: 1) (
sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt
[1]: 0) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]: 0) (
rcvTimeoutMsgCnt[0]: 1) (rcvTimeoutMsgCnt[1]: 0) (
rcvTimeoutMsgCnt[2]: 1) rcvTimeoutMsgCnt[3]: 0},'election
-timeout} {{procIds: (0, 1, 2, 3) network: (message(1, 2,
 election) message(3, 1, timeout)) tran: et(3) (proc[0]:
initiator) (proc[1]: initiator) (proc[2]: initiator) (
proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
3) (curLeader[2]: 3) (curLeader[3]: 3) (sndElectionMsgCnt
[0]: 3) (sndElectionMsgCnt[1]: 2) (sndElectionMsgCnt[2]:
1) (sndElectionMsgCnt[3]: 0) (rcvOkMsgCnt[0]: 2) (
rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]: 0) (rcvOkMsgCnt[3]:
0) (rcvTimeoutMsgCnt[0]: 1) (rcvTimeoutMsgCnt[1]: 0) (
rcvTimeoutMsgCnt[2]: 1) rcvTimeoutMsgCnt[3]: 0}, '
initiator-execution-election} {{procIds: (0, 1, 2, 3)
network: (message(2, 1, ok) message(3, 1, timeout)) tran:
 iee(2) (proc[0]: initiator) (proc[1]: initiator) (proc
[2]: initiator) (proc[3]: failedLeader) (curLeader[0]: 3)
 (curLeader[1]: 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 0) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 1) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-ok} {{
procIds: (0, 1, 2, 3) network: message(3, 1, timeout)
tran: ieo(1) (proc[0]: initiator) (proc[1]: initiator) (
proc[2]: initiator) (proc[3]: failedLeader) (curLeader
[0]: 3) (curLeader[1]: 3) (curLeader[2]: 3) (curLeader
[3]: 3) (sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]:
2) (sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
```

```
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 1) (
rcvTimeoutMsgCnt[1]: 0) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-execution-timeout} {{
procIds: (0, 1, 2, 3) network: empty tran: iet(1) (proc
[0]: initiator) (proc[1]: initiator) (proc[2]: initiator)
 (proc[3]: failedLeader) (curLeader[0]: 3) (curLeader[1]:
 3) (curLeader[2]: 3) (curLeader[3]: 3) (
sndElectionMsgCnt[0]: 3) (sndElectionMsgCnt[1]: 2) (
sndElectionMsgCnt[2]: 1) (sndElectionMsgCnt[3]: 0) (
rcvOkMsgCnt[0]: 2) (rcvOkMsgCnt[1]: 1) (rcvOkMsgCnt[2]:
0) (rcvOkMsgCnt[3]: 0) (rcvTimeoutMsgCnt[0]: 1) (
rcvTimeoutMsgCnt[1]: 1) (rcvTimeoutMsgCnt[2]: 1)
rcvTimeoutMsgCnt[3]: 0},'initiator-become-normal})
```

図 3.2 は、この反例の状態遷移を図示したものである。初期状態を S0 とし、反例が検出された状態 S34 までの遷移を示している。状態遷移で使用された書き換え規則は、観測可能成分 tran で確認できる。この反例では、状態 S34 は、tran 以外の観測可能成分が、S18 および S27 と同じ値になっており、この後は S19 から S27 までの遷移や S28 から S34 までの遷移をループすることになる。

また、弱公平性を仮定してモデル検査を行った場合も同じ反例が検出される。弱公平性では、「ラベル付きイベントが実行可能状態であり続けるとすれば、最終的にラベル付きイベントが実行される」ことを仮定できる。だが、図 3.2 からもわかるように、本稿の形式仕様を用いたモデル検査では、ラベル付きイベント ibl が実行可能状態、つまり観測可能成分 tran が bi または nee であり続けることはなく、他の書き換え規則が実行され、実行可能状態と実行不可能状態を繰り返すことになる。そのため、公平性を仮定しない場合と同様に、S19 から S27 までの遷移や S28 から S34 までの遷移をループする反例が検出される。

この問題に対応するために、強公平性を仮定して検証を行った。強公平性では「ラベル付きイベントが断続的に実行可能状態になるとすれば、最終的にラベル付きイベントが実行される」ことを仮定できる。これにより、ラベル付きイベント ibl が断続的に実行可能状態になれば、つまり観測可能成分 tran が断続的に bi または nee になれば、ラベル付きイベント ibl が実行されることになる。

なお、実験ではブリーアルゴリズムの対象となるプロセスの数を 6 以上にしてモデル検査を行ってみたが、モデル検査に時間がかかり、結果を得ることができなかった。

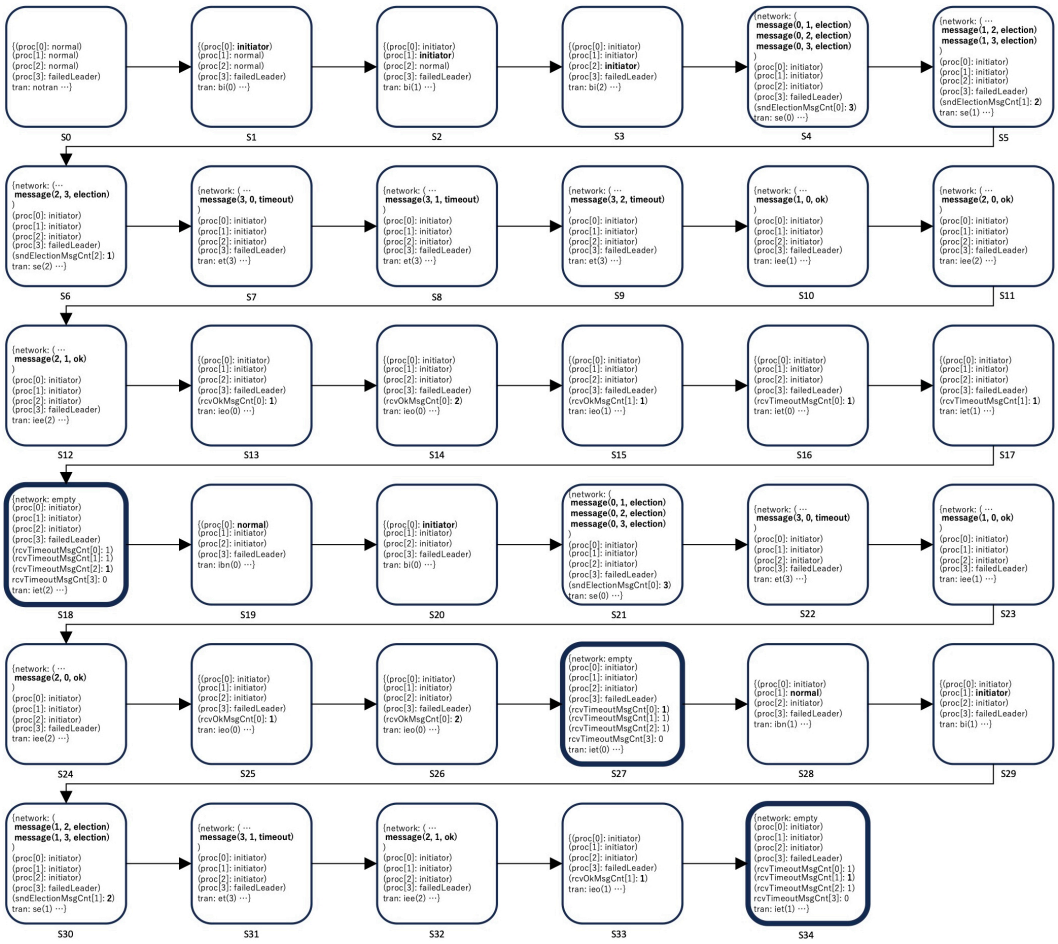

図 3.2: 公平性を仮定しないモデル検査で検出される反例の状態遷移

第4章　Chang-Robertsアルゴリズムの形式仕様とモデル検査

本章では、リーダ選出アルゴリズムの2つ目の事例としてChang-Robertsアルゴリズムを取り上げる。最初にChang-Robertsアルゴリズムの概要について説明する。次にMaudeで記述した形式仕様について説明した後、最後に作成した形式仕様を基に実施したモデル検査と結果について説明する。

4.1　Chang-Robertsアルゴリズムの概要

Chang-Robertsアルゴリズム [3, 5] は、Ernest ChangとRosemary Robertsによって提案されたリーダ選出のアルゴリズムである。このアルゴリズムでは、前提として各プロセスは、自身に与えられた比較可能な一意のIDを知っている。また、各プロセスはリング状に繋がっていて一方向の通信のみ可能とする。

最初、すべてのプロセスは非候補者であるが、リーダの不在を検知すると次の選任処理を開始する。

1. リーダの不在を検知したプロセスは、始動プロセスとして候補者となり、隣のプロセスに自身のプロセスIDを含むメッセージを送る。
2. メッセージを受け取ったプロセスが非候補者の場合は、メッセージをそのまま隣のプロセスに中継する。
3. メッセージを受け取ったプロセスが候補者の場合は、自身のプロセスIDと受信したメッセージのプロセスIDを比較して、自身のプロセスIDの方が大きい場合は、受け取ったメッセージを隣に送る。反対に、自身のプロセスIDの方が小さい場合は、受け取ったメッセージを破棄する。

上に述べた処理により、最小のプロセスIDを含むメッセージだけがリングを一周する。したがって、自身のプロセスIDを含むメッセージを受け取った場合、候補者は自身がリーダに選出されたと分かる。リーダに選出されたプロセスは、他のすべてのプロセスに自身がリーダとなることを知らせるため、自身のプロセスIDを含むメッセージをもう一度、隣のプロセスに送る。選出されたプロセスが、

送ったメッセージを受け取るとプロセスはリーダとなり、アルゴリズムは終了となる。

図 4.1 に、Chang-Roberts アルゴリズムがどのように動作するかを示す。すべてのプロセスはリング状に配置され、右回りにメッセージを送ることができる。図 4.1(a) では、プロセス 6、4、7、2 がリーダの不在を検知し、始動プロセスとして候補者となっている。また、候補者となったプロセスは、それぞれ自身のプロセス ID を含む Candidate メッセージを隣のプロセスに送っている。プロセス 6 が送った Candidate メッセージは、隣の非候補者のプロセスが受け取り中継した後、プロセス 4 が受け取る。プロセス 4 は、自身のプロセス ID の方が Candidate メッセージのプロセス ID より小さいため、プロセス 6 からの Candidate メッセージを破棄する。同様に、プロセス 4 およびプロセス 7 が送った Candidate メッセージは、いくつかのプロセスが中継した後、プロセス 2 が受け取る。プロセス 2 は、自身のプロセス ID の方が Candidate メッセージのプロセス ID より小さいため、Candidate をメッセージを破棄する。プロセス 2 が送った Candidate メッセージは、候補者の中で最小のプロセス ID であるため、他の候補者プロセスに破棄されることなくリングを一周してプロセス 2 に戻る。プロセス 2 は、自身の送った Candidate メッセージが、一周したため、自身がリーダになることを知る。図 4.1(b) では、新たにリーダとなるプロセス 2 が、自身のプロセス ID を含む Coordinator メッセージを送り、他のすべてのプロセスに新しいリーダのプロセス ID を知らせる。プロセス 2 が、リングを一周した Coordinator メッセージを受け取り終了となる。

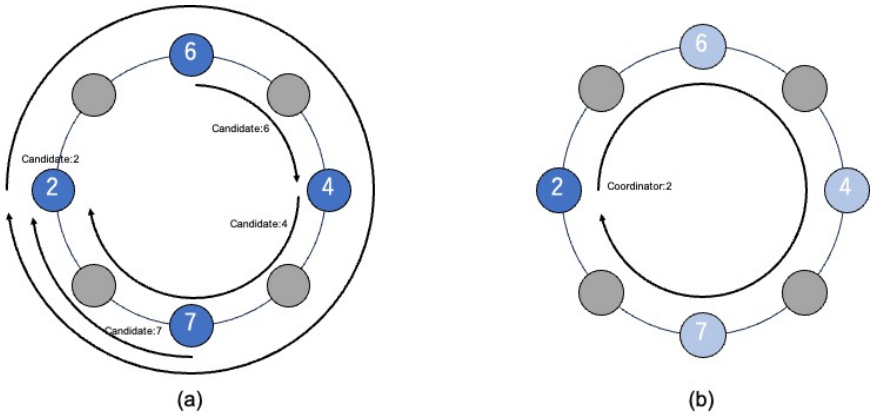

図 4.1: Chang-Roberts アルゴリズム

4.2 Chang-Roberts アルゴリズムの形式仕様

4.2.1 観測可能成分

Chang-Roberts アルゴリズムでは、ソースコード 4.1 に示す 4 つの観測可能成分を用いて状態遷移システムを形式化した。

ソースコード 4.1: Chang-Roberts アルゴリズムの観測可能成分

```
1 op proc[_]:_ : Nat Process -> OComp [ctor] .
2 op rcvCandidateMsgCnt[_]:_ : Nat Nat -> OComp [ctor] .
3 op rcvCoordinatorMsgCnt[_]:_ : Nat Nat -> OComp [ctor] .
4 op network:_ : Soup{Message} -> OComp [ctor] .
```

`Process` はプロセス、`OComp` は観測可能成分、`Message` はメッセージをそれぞれ表すソートである。

`proc[_]:_` は、第 1 引数にプロセス ID、第 2 引数にプロセスをとる。プロセスは、`normal`、`cand`、`lost`、`elected`、`leader` のいずれかの状態、リーダと認識しているプロセスの ID および隣のプロセスの ID を保持している。この観測可能成分は、プロセスを表しており、第 1 引数のプロセス ID を持つプロセスが、第 2 引数のプロセスであることを意味する。

`rcvCandidateMsgCnt[_]:_` は、第 1 引数にプロセス ID、第 2 引数にプロセスが受信した Candidate メッセージの回数をとる。この観測可能成分は、選任で自身が送信した Candidate メッセージを受信した回数を表しており、第 1 引数のプロセス ID を持つプロセスが、選任で自身が送信した Candidate メッセージを受信した回数は、第 2 引数の値であることを意味する。

`rcvCoordinatorMsgCnt[_]:_` は、第 1 引数にプロセス ID、第 2 引数にプロセスが受信した Coordinator メッセージの回数をとる。この観測可能成分は、選任で自身が送信した Coordinator メッセージを受信した回数を表しており、第 1 引数のプロセス ID を持つプロセスが、選任で自身が送信した Coordinator メッセージを受信した回数は、第 2 引数の値であることを意味する。

`network:_` は、引数にメッセージのスープをとる。この観測可能成分は、各プロセスが選任処理で送受信するメッセージのためのネットワークを表し、引数のメッセージがネットワーク内にあるを意味する。

Chang-Roberts アルゴリズムでは、各プロセスはリング状に配置されているが、本稿の形式仕様ではネットワークトポロジのモデル化は行わない。各プロセスに隣のプロセス ID を保持させ、そのプロセス ID をメッセージの送信先に設定することで、ネットワークノード間のリンクを表現し、各プロセスが処理対象となるメッセージを特定可能にしている。

4.2.2 書き換え規則

Chang-Roberts アルゴリズムを形式仕様として表現するために、以降に示す 9 つの書き換え規則を利用する。また、これらの書き換え規則では表 4.1 の変数を用いている。

表 4.1: Chang-Roberts アルゴリズムの書き換え規則で用いる変数

変数名	説明
S	観測可能成分の集合の変数
ID、LID、NID、CID、EID	プロセス ID の変数
RC、RC0、RC1	受信したメッセージ回数の変数
NW	メッセージの集合の変数

ソースコード 4.2: start-election の定義

```
1 rl [start-election] :
2  {(proc[ID]: process(normal, LID, NID))
3   (rcvCandidateMsgCnt[ID]: RC) (network: NW) S} =>
4  {(proc[ID]: process(cand, LID, NID))
5   (rcvCandidateMsgCnt[ID]: 0)
6   (network: (message(NID, candidate, ID) NW)) S} .
```

ソースコード 4.2 は、書き換え規則 start-election の定義である。この書き換え規則は、リーダの不在を検知した normal 状態のプロセス ID が、始動プロセスとして cand 状態（候補者）になり、選任を開始することを表している。cand 状態になったプロセス ID は、隣のプロセス NID に candidate メッセージ message(NID, candidate, ID) を送信する。また、プロセス ID が受信した candidate メッセージ回数を初期化するため、(rcvCandidateMsgCnt[ID]: 0) としている。

ソースコード 4.3: normal-execution の定義

```
1 rl [normal-execution] :
2 {(proc[ID]: process(normal, LID, NID))
3  (network: (message(ID, candidate, CID) NW)) S} =>
4 {(proc[ID]: process(lost, LID, NID))
5  (network: (message(NID, candidate, CID) NW)) S} .
```

ソースコード 4.3 は、書き換え規則 normal-execution の定義である。この書き換え規則は、normal 状態のプロセス ID が、candidate メッセージ message(ID, candidate, CID) を受信した場合、lost 状態になることを表している。また、プロセス ID は非候補者であるため、受信した candidate メッセージは、そのまま隣のプロセス NID に message(NID, candidate, CID) として送信している。

ソースコード 4.4: cand-execution-ignore の定義

```
1 crl [cand-execution-ignore] :
```

```
 {(proc[ID]: process(cand, LID, NID))
  (network: (message(ID, candidate, CID) NW)) S} =>
 {(proc[ID]: process(cand, LID, NID)) (network: NW) S}
 if ID < CID .
```

ソースコード 4.4 は、書き換え規則 cand-execution-ignore の定義である。この書き換え規則は、cand 状態のプロセス ID が、自身のプロセス ID より大きなプロセス ID であるプロセス CID の candidate メッセージ message(ID, candidate, CID) を受信した場合、その candidate メッセージを破棄することを表している。

ソースコード 4.5: cand-execution-lost の定義

```
crl [cand-execution-lost] :
 {(proc[ID]: process(cand, LID, NID))
  (network: (message(ID, candidate, CID) NW)) S} =>
 {(proc[ID]: process(lost, LID, NID))
  (network: (message(NID, candidate, CID) NW)) S}
 if ID > CID .
```

ソースコード 4.5 は、書き換え規則 cand-execution-lost の定義である。この書き換え規則は、cand 状態のプロセス ID が、自身のプロセス ID より小さなプロセス ID であるプロセス CID の candidate メッセージ message(ID, candidate, CID) を受信した場合、lost 状態になることを表している。また、受信した candidate メッセージは、隣のプロセス NID に message(NID, candidate, CID) として送信している。

ソースコード 4.6: cand-execution-elected の定義

```
crl [cand-execution-elected] :
{(proc[ID]: process(cand, LID, NID))
 (rcvCandidateMsgCnt[ID]: RC0)
 (rcvCoordinatorMsgCnt[ID]: RC1)
 (network: (message(ID, candidate, CID) NW)) S} =>
{(proc[ID]: process(elected, LID, NID))
 (rcvCandidateMsgCnt[ID]: RC0 + 1)
 (rcvCoordinatorMsgCnt[ID]: 0)
 (network: (message(NID, coordinator, ID) NW)) S}
if ID == CID .
```

ソースコード 4.6 は、書き換え規則 cand-execution-elected の定義である。この書き換え規則は、cand 状態のプロセス ID が、自身のプロセス ID と同じプロセス ID であるプロセス CID の candidate メッセージ message(ID, candidate, CID) を受信した場合、elected 状態になることを表している。elected 状態になったプロセス ID は、隣のプロセス NID に coordinator メッセージ message(NID, coordinator, ID) を送信する。また、プロセス ID が受信した candidate メッセージ回数をインクリメントするため、(rcvCandidateMsgCnt[ID]: RC0 + 1) とし、受信した coordinator メッセージ回数を初期化するため、(rcvCoordinatorMsgCnt[ID]: 0)

としている。Chang-Roberts アルゴリズムでは、各候補者プロセスが送信した candidate メッセージの中で、最小のプロセス ID を含む candidate メッセージだけが、リングを一周する。自身のプロセス ID と同じプロセス ID を含む candidate メッセージを受信したことは、candidate メッセージがリングを一周したことを示唆している。そのため、この書き換え規則では、プロセス ID を選任処理により選出されたプロセスとして elected 状態にしている。

ソースコード 4.7: elected-execution の定義

```
crl [elected-execution] :
 {(proc[ID]: process(elected, LID, NID))
  (rcvCoordinatorMsgCnt[ID]: RC)
  (network: (message(ID, coordinator, EID) NW)) S} =>
 {(proc[ID]: process(leader, EID, NID))
  (rcvCoordinatorMsgCnt[ID]: RC + 1) (network: NW) S}
if ID == EID .
```

ソースコード 4.7 は、書き換え規則 elected-execution の定義である。この書き換え規則は、elected 状態のプロセス ID が、自身のプロセス ID と同じプロセス ID であるプロセス EID の coordinator メッセージ message(ID, coordinator, EID) を受信した場合、leader 状態になることを表している。また、プロセス ID が、リーダと認識しているプロセスの ID を EID に変更し、受信した coordinator メッセージ回数をインクリメントするため、(rcvCoordinatorMsgCnt[ID]: RC + 1) としている。自身のプロセス ID と同じプロセス ID を含む coordinator メッセージを受信したことは、coordinator メッセージがリングを一周したことを示唆している。そのため、この書き換え規則では、coordinator メッセージによる他のすべてのプロセスへの通知は終了したものとし、プロセス ID を leader 状態にしている。

ソースコード 4.8: lost-recieve-cand-msg の定義

```
rl [lost-recieve-cand-msg] :
 {(proc[ID]: process(lost, LID, NID))
  (network: (message(ID, candidate, CID) NW)) S} =>
 {(proc[ID]: process(lost, LID, NID))
  (network: (message(NID, candidate, CID) NW)) S} .
```

ソースコード 4.8 は、書き換え規則 lost-recieve-cand-msg の定義である。この書き換え規則は、lost 状態のプロセス ID が、candidate メッセージ message(ID, candidate, CID) を受信した場合、その candidate メッセージをそのまま隣のプロセス NID に message(NID, candidate, CID) として送信することを表している。

ソースコード 4.9: lost-recieve-coord-msg の定義

```
rl [lost-recieve-coord-msg] :
 {(proc[ID]: process(lost, LID, NID))
  (network: (message(ID, coordinator, EID) NW)) S} =>
 {(proc[ID]: process(lost, EID, NID))
  (network: (message(NID, coordinator, EID) NW)) S} .
```

ソースコード 4.9 は、書き換え規則 lost-recieve-coord-msg の定義である。この書き換え規則は、lost 状態のプロセス ID が、coordinator メッセージ message(ID, coordinator, EID) を受信した場合、プロセス ID が、リーダと認識しているプロセス ID を EID に変更することを表している。また、プロセス ID は、受信した coordinator メッセージをそのまま隣のプロセス NID に message(NID, coordinator, EID) として送信している。

ソースコード 4.10: leader-recieve-cand-msg の定義

```
1  rl [leader-recieve-cand-msg] :
2   {(proc[ID]: process(leader, LID, NID))
3    (network: (message(ID, candidate, CID) NW)) S} =>
4   {(proc[ID]: process(leader, LID, NID))
5    (network: (message(NID, candidate, CID) NW)) S} .
```

ソースコード 4.10 は、書き換え規則 leader-recieve-cand-msg の定義である。この書き換え規則は、leader 状態のプロセス ID が、candidate メッセージ message(ID, candidate, CID) を受信した場合、その candidate メッセージをそのまま隣のプロセス NID に message(NID, candidate, CID) として送信することを表している。

4.2.3 LTL 式

Chang-Roberts アルゴリズムが、活性を満たすことのモデル検査を行うために、以降に示す LTL 式を利用する。

ソースコード 4.11: CR-PREDS モジュールの定義

```
1  mod CR-PREDS is
2   pr CHANG-ROBERTS .
3   inc SATISFACTION .
4   subsort CRState < State .
5   ops cand elected leader : Nat -> Prop .
6   op rcvMyCandidateMsg : Nat -> Prop .
7   op rcvMyCoordinatorMsg : Nat -> Prop .
8   vars ID LID NID : Nat .
9   var S : Soup{OComp} .
10  var PROP : Prop .
11  eq {(proc[ID]: process(cand, LID, NID)) S}
12   |= cand(ID) = true .
13  eq {(proc[ID]: process(elected, LID, NID)) S}
14   |= elected(ID) = true .
15  eq {(proc[ID]: process(leader, LID, NID)) S}
16   |= leader(ID) = true .
17  eq {(rcvCandidateMsgCnt[ID]: 1) S}
18   |= rcvMyCandidateMsg(ID) = true .
```

```
 eq {(rcvCoordinatorMsgCnt[ID]: 1) S}
  |= rcvMyCoordinatorMsg(ID) = true .
 eq {S} |= PROP = false [owise] .
endm
```

ソースコード 4.11 は、CR-PREDS モジュールの定義である。このモジュールは、Chang-Roberts アルゴリズムの性質を LTL 式で表現するために必要な原子命題を定義している。

CR-PREDS は、CHANG-ROBERTS および SATISFACTION をインポートしている。CHANG-ROBERTS は、4.2.2 項で説明した書き換え規則を定義したモジュールである。また、SATISFACTION は、Maude で LTL モデル検査器を利用するために必要な model-checker.maude に定義されているモジュールで、状態と原子命題の充足関係を表す|=演算子が記述されている。CR-PREDS では、|=演算子を利用して、LTL 式に必要な状態と原子命題の充足関係を定義している。例えば、状態が{(proc[ID]: process(leader, LID, NID)) S}の場合に、原子命題 leader(ID) は充足する。

ソースコード 4.12: CR-CHECK モジュールの定義

```
mod CR-CHECK is
 inc CR-PREDS .
 inc MODEL-CHECKER .
 inc LTL-SIMPLIFIER .
 op leaderLiveness : -> Formula .
 op beLeader : -> Formula .
 op candidateMsgCircleRing : -> Formula .
 op coordinatorMsgCircleRing : -> Formula .
 eq leaderLiveness =
  <> (leader(0) \/ leader(1) \/ leader(2) \/
   leader(3) \/ leader(4)) .
 eq beLeader =
  (cand(0) |-> leader(0)) \/ (cand(1) |-> leader(1)) \/
  (cand(2) |-> leader(2)) \/ (cand(3) |-> leader(3)) \/
  (cand(4) |-> leader(4)) .
 eq candidateMsgCircleRing =
  (cand(0) |-> rcvMyCandidateMsg(0)) \/
  (cand(1) |-> rcvMyCandidateMsg(1)) \/
  (cand(2) |-> rcvMyCandidateMsg(2)) \/
  (cand(3) |-> rcvMyCandidateMsg(3)) \/
  (cand(4) |-> rcvMyCandidateMsg(4)) .
 eq coordinatorMsgCircleRing =
  ((cand(0) |-> elected(0)) /\
   (elected(0) |-> rcvMyCoordinatorMsg(0))) \/
  ((cand(1) |-> elected(1)) /\
   (elected(1) |-> rcvMyCoordinatorMsg(1))) \/
  ((cand(2) |-> elected(2)) /\
   (elected(2) |-> rcvMyCoordinatorMsg(2))) \/
  ((cand(3) |-> elected(3)) /\
```

```
30     (elected(3) |-> rcvMyCoordinatorMsg(3))) \/
31   ((cand(4) |-> elected(4)) /\
32     (elected(4) |-> rcvMyCoordinatorMsg(4))) .
33 endm
```

ソースコード 4.12 は、CR-CHECK モジュールの定義である。このモジュールは、モデル検査用のモジュールで Chang-Roberts アルゴリズムの性質を表す LTL 式を定義している。

CR-CHECK は、CR-PREDS、MODEL-CHECKER および LTL-SIMPLIFIER をインポートしている。MODEL-CHECKER および LTL-SIMPLIFIER は、model-checker.maude に定義されているモジュールである。MODEL-CHECKER には、LTL モデル検査で利用する modelCheck 関数が記述されている。LTL-SIMPLIFIER には、LTL モデル検査のための補助モジュールで、LTL 式を簡略化する等式が定義されている。

CR-CHECK では、LTL 式を 4 つを定義している。1 つ目の leaderLiveness は、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質を表す LTL 式である。2 つ目の beLeader は、「リーダの候補者になれば、いずれリーダになる」という活性の性質を表す LTL 式である。3 つ目の candidateMsgCircleRing は、「リーダの候補者になれば、いずれ自身が送信した candidate メッセージはリングを一周する」という活性の性質を表す LTL 式である。4 つ目は coordinatorMsgCircleRing は、「リーダに選出されれば、いずれ自身が送信した coordinator メッセージはリングを一周する」という活性の性質を表す LTL 式である。

4.3 Chang-Roberts アルゴリズムのモデル検査

本節では、前節で説明した形式仕様に基づいて、Chang-Roberts アルゴリズムの正当性とアルゴリズム固有の性質が満たされているかをモデル検査で検証する。

アルゴリズムの正当性では次の 2 つの性質を検証する。

- 「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質
- 「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質

アルゴリズム固有の性質では次の 3 つの性質を検証する。

- 「リーダの候補者になれば、いずれリーダになる」という活性の性質
- 「リーダの候補者になれば、いずれ自身が送信した candidate メッセージはリングを一周する」という活性の性質
- 「リーダに選出されれば、いずれ自身が送信した coordinator メッセージはリングを一周する」という活性の性質

モデル検査を実施するにあたり、プロセスの並び順を考慮した初期状態を 2 つ定義する。1 つ目は、プロセスがプロセス ID の昇順にリング状に配置されている初期状態 init である。2 つ目は、プロセスがランダムにリング状に配置されている初期状態 init-random である。Chang-Roberts アルゴリズムの対象になるプロセスが 5 つで、それぞれのプロセス ID が 0 から 4 の場合、初期状態 init および初期状態 init-random を下記のように定義する。

ソースコード 4.13: 初期状態 init の定義

```
op {_} : Soup{OComp} -> CRState [ctor] .
op init : -> CRState .
eq init = {
 (proc[0]: process(normal, 0, 1))
 (proc[1]: process(normal, 1, 2))
 (proc[2]: process(normal, 2, 3))
 (proc[3]: process(normal, 3, 4))
 (proc[4]: process(normal, 4, 0))
 (rcvCandidateMsgCnt[0]: 0) (rcvCandidateMsgCnt[1]: 0)
 (rcvCandidateMsgCnt[2]: 0) (rcvCandidateMsgCnt[3]: 0)
 (rcvCandidateMsgCnt[4]: 0)
 (rcvCoordinatorMsgCnt[0]: 0) (rcvCoordinatorMsgCnt[1]: 0)
 (rcvCoordinatorMsgCnt[2]: 0) (rcvCoordinatorMsgCnt[3]: 0)
 (rcvCoordinatorMsgCnt[4]: 0)
 (network: empty)
} .
```

ソースコード 4.14: 初期状態 init-random の定義

```
op init-random : -> CRState .
eq init-random = {
 (proc[0]: process(normal, 0, 3))
 (proc[1]: process(normal, 1, 4))
 (proc[2]: process(normal, 2, 0))
 (proc[3]: process(normal, 3, 1))
 (proc[4]: process(normal, 4, 2))
 (rcvCandidateMsgCnt[0]: 0) (rcvCandidateMsgCnt[1]: 0)
 (rcvCandidateMsgCnt[2]: 0) (rcvCandidateMsgCnt[3]: 0)
 (rcvCandidateMsgCnt[4]: 0)
 (rcvCoordinatorMsgCnt[0]: 0) (rcvCoordinatorMsgCnt[1]: 0)
 (rcvCoordinatorMsgCnt[2]: 0) (rcvCoordinatorMsgCnt[3]: 0)
 (rcvCoordinatorMsgCnt[4]: 0)
 (network: empty)
} .
```

ソースコード 4.13 は初期状態 init、ソースコード 4.14 は初期状態 init-random の定義である。各プロセスは、初期状態 init ではプロセス ID の昇順に、初期状態 init-random では 3、1、4、2、0 の順に配置されている。プロセスの状態は、両初期状態ともすべてのプロセスを normal 状態とし、リーダと認識しているプロセ

スは自身のプロセスとする。選任で自身が送信した Candidate メッセージおよび Coordinator メッセージを受信した回数については 0 とする。ネットワークワークは empty とし、ネットワークにメッセージはないものとする。

まず、Chang-Roberts アルゴリズムの正当性を検証する。「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質は、ソースコード 4.15 のコマンドを、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質は、ソースコード 4.16 のコマンドをそれぞれ実行することで確認できる。

ソースコード 4.15: 正当性における安全性の性質を確認するコマンド

```
1 search [1] in CHANG-ROBERTS : init =>*
2  {(proc[ID0]: process(leader, LID0, NID0))
3   (proc[ID1]: process(leader, LID1, NID1)) S} .
4 search [1] in CHANG-ROBERTS : init-random =>*
5  {(proc[ID0]: process(leader, LID0, NID0))
6   (proc[ID1]: process(leader, LID1, NID1)) S} .
```

ソースコード 4.16: 正当性における活性の性質を確認するコマンド

```
1 red in CR-CHECK : modelCheck(init, leaderLiveness) .
2 red in CR-CHECK : modelCheck(init-random, leaderLiveness) .
```

ソースコード 4.15 およびソースコード 4.16 のコマンドを実行して、Maude から得られる結果は以下の通りである。

```
search [1] in CHANG-ROBERTS : init =>* {S (proc[ID0]:
   process(leader, LID0, NID0)) proc[ID1]: process(leader,
   LID1, NID1)} .

No solution.
states: 4080  rewrites: 20418 in 102ms cpu (102ms real)
   (200082 rewrites/second)
==========================================
search [1] in CHANG-ROBERTS : init-random =>* {S (proc[ID0]:
    process(leader, LID0, NID0)) proc[ID1]: process(leader,
   LID1, NID1)} .

No solution.
states: 3462  rewrites: 16001 in 69ms cpu (69ms real)
   (229362 rewrites/second)
==========================================
reduce in CR-CHECK : modelCheck(init, leaderLiveness) .
rewrites: 29043 in 59ms cpu (59ms real) (489376 rewrites/
   second)
result Bool: true
==========================================
reduce in CR-CHECK : modelCheck(init-random, leaderLiveness)
    .
```

```
rewrites: 21398 in 43ms cpu (43ms real) (494145 rewrites/
   second)
result Bool: true
```

ソースコード 4.15 のコマンドの実行結果は、初期状態 init および初期状態 init-random から、2 つの異なるプロセスが同時にリーダになる状態への遷移は見つからなかったことを示している。これにより、「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質が満たされることが分かった。次に、ソースコード 4.16 のコマンドの実行結果は、反例は検出されず、モデル検査が成功している。これにより、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質が満たされることが分かった。これらの結果から、Chang-Roberts アルゴリズムの正当性の性質は満たされるいる。

次に、Chang-Roberts アルゴリズム固有の性質を検証する。「リーダの候補者になれば、いずれリーダになる」という活性の性質は、ソースコード 4.15 のコマンドを、「リーダの候補者になれば、いずれ自身が送信した candidate メッセージはリングを一周する」という活性の性質は、ソースコード 4.16 のコマンドを、「リーダに選出されれば、いずれ自身が送信した coordinator メッセージはリングを一周する」という活性の性質は、ソースコード 4.16 のコマンドをそれぞれ実行することで確認できる。

ソースコード 4.17: LTL 式 beLeader を確認するコマンド

```
1 red in CR-CHECK : modelCheck(init, beLeader) .
2 red in CR-CHECK : modelCheck(init-random, beLeader) .
```

ソースコード 4.18: LTL 式 candidateMsgCircleRing を確認するコマンド

```
1 red in CR-CHECK : modelCheck(init, candidateMsgCircleRing) .
2 red in CR-CHECK :
3  modelCheck(init-random, candidateMsgCircleRing) .
```

ソースコード 4.19: LTL 式 coordinatorMsgCircleRing を確認するコマンド

```
1 red in CR-CHECK :
2  modelCheck(init, coordinatorMsgCircleRing) .
3 red in CR-CHECK :
4  modelCheck(init-random, coordinatorMsgCircleRing) .
```

ソースコード 4.17、ソースコード 4.18 およびソースコード 4.19 のコマンドを実行して、Maude から得られる結果は以下の通りである。

```
reduce in CR-CHECK : modelCheck(init, beLeader) .
rewrites: 58932 in 272ms cpu (272ms real) (216454 rewrites/
   second)
result Bool: true
==========================================
reduce in CR-CHECK : modelCheck(init-random, beLeader) .
```

```
rewrites: 48894 in 207ms cpu (207ms real) (235935 rewrites/
   second)
result Bool: true
==========================================
reduce in CR-CHECK : modelCheck(init, candidateMsgCircleRing
   ) .
rewrites: 57902 in 239ms cpu (239ms real) (241673 rewrites/
   second)
result Bool: true
==========================================
reduce in CR-CHECK : modelCheck(init-random,
   candidateMsgCircleRing) .
rewrites: 48029 in 193ms cpu (193ms real) (247991 rewrites/
   second)
result Bool: true
==========================================
reduce in CR-CHECK : modelCheck(init,
   coordinatorMsgCircleRing) .
rewrites: 69130 in 765ms cpu (766ms real) (90290 rewrites/
   second)
result Bool: true
==========================================
reduce in CR-CHECK : modelCheck(init-random,
   coordinatorMsgCircleRing) .
rewrites: 58335 in 631ms cpu (632ms real) (92403 rewrites/
   second)
result Bool: true
```

最初に、ソースコード 4.17 のコマンドの実行結果では、反例は検出されず、モデル検査が成功している。これにより、「リーダの候補者になれば、いずれリーダになる」という活性の性質が満たされることが分かる。次に、ソースコード 4.18 のコマンドの実行結果でも反例は検出されず、モデル検査が成功している。これにより、「リーダの候補者になれば、いずれ自身が送信した candidate メッセージはリングを一周する」という活性の性質も満たされることが分かる。最後に、ソースコード 4.19 のコマンドの実行結果も他の 2 つの結果と同じく反例は検出されず、モデル検査が成功している。これにより、「リーダに選出されれば、いずれ自身が送信した coordinator メッセージはリングを一周する」という活性の性質も満たされることが分かる。

4.4 まとめ

本章では、Chang-Roberts アルゴリズムの形式仕様を作成し、モデル検査を行った。モデル検査の実験では、Chang-Roberts アルゴリズムの対象となるプロセスを

5つとし、プロセスの並び順を考慮した2つの初期状態において、Chang-Robertsアルゴリズムの正当性とアルゴリズム固有の性質が満たされていることを検証した。

アルゴリズムの正当性では、「常に2つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質と「いつか少なくとも1つのプロセスが、リーダに選ばれる」という活性の性質を検証し、2つの性質が同時に満たされていることを検証した。これにより、Chang-Robertsアルゴリズムは、リーダ選出アルゴリズムとしての正当性を満たしていることを確認できた。

アルゴリズム固有の性質では、「リーダの候補者になれば、いずれリーダになる」という活性の性質、「リーダの候補者になれば、いずれ自身が送信したcandidateメッセージはリングを一周する」という活性の性質、および「リーダに選出されれば、いずれ自身が送信したcoordinatorメッセージはリングを一周する」という活性の性質が、それぞれ満たされていることを確認できた。

なお、実験ではChang-Robertsアルゴリズムの対象となるプロセスの数を6以上にしてモデル検査を行ってみたが、初期状態のプロセスが8以上になると、モデル検査に時間がかかり、結果を得ることができなかった。

第5章　Franklinアルゴリズムの形式仕様とモデル検査

本章では、リーダ選出アルゴリズムの最後の事例としてFranklinアルゴリズムを取り上げる。最初にFranklinアルゴリズムの概要について説明する。次にMaudeで記述した形式仕様について説明した後、最後に作成した形式仕様を基に実施したモデル検査と結果について説明する。

5.1　Franklinアルゴリズムの概要

Franklinアルゴリズム [5] は、W. Randolph Franklinによって提案されたリーダ選出のアルゴリズムで、Chang-Robertsアルゴリズムのメッセージの複雑性を改善することを目的としている。このアルゴリズムでは、前提として各プロセスは、自身に与えられた比較可能な一意のIDを知っている。また、各プロセスはリング状に繋がっていて、双方向の通信が可能である。

最初、すべてのプロセスは非イニシエーターであるが、リーダの不在を検知すると次の選任処理を開始する。

1. リーダの不在を検知したプロセスは、アクティブプロセスとしてイニシエーターとなり、両隣のプロセスに自身のプロセスIDを含むメッセージを送る。
2. メッセージを受け取ったプロセスが非イニシエーターの場合は、パッシブプロセスとしてメッセージをそのまま隣のプロセスに中継する。
3. メッセージを受け取ったプロセスがイニシエーターの場合は、両隣のプロセスから受信したメッセージのプロセスIDのうち、大きな方のプロセスIDと自身のプロセスIDを比較する。自身のプロセスIDの方が大きい場合は、再度、両隣のプロセスに自身のプロセスIDを含むメッセージを送る。自身のプロセスIDの方が小さい場合は、パッシブプロセスとなる。自身のプロセスIDと等しい場合は、リーダとなる。

上に述べた処理によりリーダが選出される。リーダに選出されたプロセスは、他のすべてのプロセスに自身がリーダとなることを知らせるため、自身のプロセスIDを含むメッセージを隣のプロセスに送る。送ったメッセージがリングを一周

し、リーダに選出されたプロセスがメッセージを受け取るとアルゴリズムは終了となる。

図 5.1 に、Franklin アルゴリズムがどのように動作するかを示す。すべてのプロセスはリング状に配置され、双方向にメッセージを送ることができる。図 5.1(a) では、プロセス 4、3、5 がリーダの不在を検知し、アクティブプロセスとしてイニシエーターになっている。また、イニシエーターとなったプロセスは、それぞれ自身のプロセス ID を含む Election メッセージを両隣のプロセスに送っている。

図 5.1(b) では、プロセス 4 が、プロセス 3 とプロセス 5 が送った Election メッセージを受け取り、大きな方のプロセス ID である 5 と自身のプロセス ID である 4 を比較する。自身のプロセス ID の方が小さいため、プロセス 4 はパッシブプロセスとなる。同様にプロセス 3 も、プロセス 4 とプロセス 5 が送った Election メッセージを受け取り、大きな方のプロセス ID である 5 と自身のプロセス ID である 3 を比較する。プロセス 3 も自身のプロセス ID の方が小さいため、パッシブプロセスとなる。プロセス 5 は、プロセス 3 とプロセス 4 が送った Election メッセージを受け取り、大きな方のプロセス ID である 4 と自身のプロセス ID である 5 を比較する。プロセス 5 は、自身のプロセス ID の方が大きいため、再度、両隣に自身のプロセス ID を含む Election メッセージを送っている。

図 5.1(c) では、プロセス 5 が、図 5.1(b) で再送した Election メッセージを自身で受け取る。受け取ったプロセス ID と自身のプロセス ID を比較した結果、共に 5 で等しくなるため、プロセス 5 がリーダに選出される。リーダに選出されたプロセス 5 は、他のすべてのプロセスにリーダとなることを知らせるため、自身のプロセス ID を含む Elected メッセージを右回りで送っている。プロセス 5 が、リングを一周した Elected メッセージを受け取り終了となる。

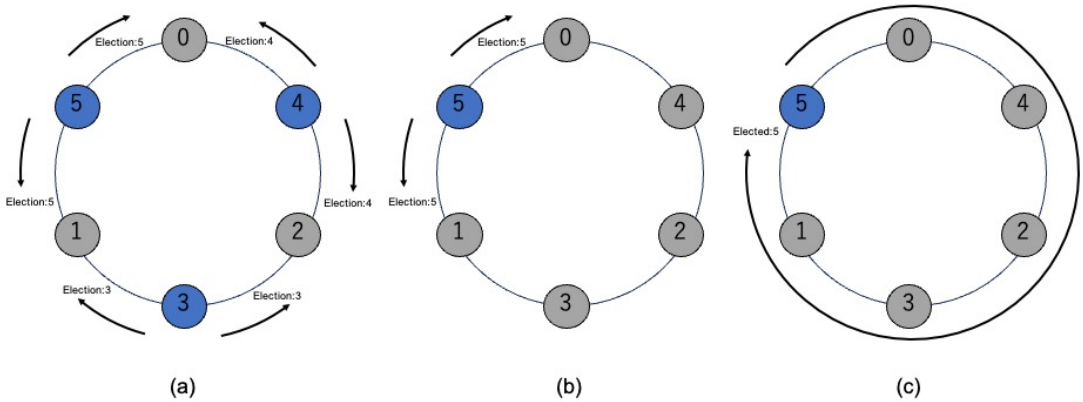

図 5.1: Franklin アルゴリズム

5.2 Franklin アルゴリズムの形式仕様

5.2.1 観測可能成分

Franklin アルゴリズムでは、ソースコード 5.1 に示す 4 つの観測可能成分を用いて状態遷移システムを形式化した。

ソースコード 5.1: Franklin アルゴリズムの観測可能成分

```
1 op proc[_]:_ : Nat Process -> OComp [ctor] .
2 op rcvLeft[_]:_ : Nat MessageType -> OComp [ctor] .
3 op rcvRight[_]:_ : Nat MessageType -> OComp [ctor] .
4 op network:_ : Soup{Message} -> OComp [ctor] .
```

`Process` はプロセス、`OComp` は観測可能成分、`MessageType` はメッセージ種別、`Message` はメッセージをそれぞれ表すソートである。

`proc[_]:_` は、第 1 引数にプロセス ID、第 2 引数にプロセスをとる。プロセスは、`leader`、`normal`、`initiator`、`passive` のいずれかの状態、リーダと認識しているプロセスの ID および両隣のプロセスの ID を保持している。この観測可能成分は、プロセスを表しており、第 1 引数のプロセス ID を持つプロセスが、第 2 引数のプロセスであることを意味する。

`rcvLeft[_]:_` は、第 1 引数にプロセス ID、第 2 引数にメッセージ種別をとる。この観測可能成分は、選任で左側に配置されているプロセスから受け取ったメッセージ種別を表しており、第 1 引数のプロセス ID を持つプロセスが、選任で左側に配置されているプロセスから受け取ったメッセージ種別は、第 2 引数のメッセージ種別であることを意味する。

`rcvRight[_]:_` は、第 1 引数にプロセス ID、第 2 引数にメッセージ種別をとる。この観測可能成分は、選任で右側に配置されているプロセスから受け取ったメッセージ種別を表しており、第 1 引数のプロセス ID を持つプロセスが、選任で右側に配置されているプロセスから受け取ったメッセージ種別は、第 2 引数のメッセージ種別であることを意味する。

`network:_` は、引数にメッセージのスープをとる。この観測可能成分は、各プロセスが選任処理で送受信するメッセージのためのネットワークを表し、引数のメッセージがネットワーク内にあるを意味する。

Franklin アルゴリズムでは、各プロセスはリング状に配置されており、双方向の通信が可能であるが、本稿の形式仕様ではネットワークトポロジのモデル化は行わない。各プロセスに両隣のプロセス ID を保持させ、そのプロセス ID をメッセージの送受信先に設定することで、ネットワークノード間のリンクを表現し、各プロセスが処理対象となるメッセージを特定できるようにしている。

5.2.2 書き換え規則

Franklin アルゴリズムを形式仕様として表現するために、以降に示す 12 の書き換え規則を利用する。また、これらの書き換え規則では表 5.1 の変数を用いている。

表 5.1: Franklin アルゴリズムの書き換え規則で用いる変数

変数名	説明
S	観測可能成分の集合の変数
ID、ID0、ID1、LID、EID、ID_L、ID_R	プロセス ID の変数
NW	メッセージの集合の変数

ソースコード 5.2: start-election の定義

```
1 rl [start-election] :
2  {(proc[ID]: process(normal, LID, ID_L, ID_R))
3   (rcvLeft[ID]: none) (rcvRight[ID]: none)
4   (network: NW) S} =>
5  {(proc[ID]: process(initiator, LID, ID_L, ID_R))
6   (rcvLeft[ID]: none) (rcvRight[ID]: none)
7   (network: (
8     message(election(ID), ID, ID_L)
9     message(election(ID), ID, ID_R) NW)) S} .
```

ソースコード 5.2 は、書き換え規則 start-election の定義である。この書き換え規則は、リーダの不在を検知した normal 状態のプロセス ID が、始動プロセスとして initiator 状態になり、選任を開始することを表している。initiator 状態になったプロセス ID は、両隣のプロセス ID_L および ID_R に Election メッセージ message(election(ID), ID, ID_L) と message(election(ID), ID, ID_R) を送信する。

ソースコード 5.3: initiator-rcv-left の定義

```
1 rl [initiator-rcv-left] :
2  {(proc[ID]: process(initiator, LID, ID_L, ID_R))
3   (rcvLeft[ID]: none)
4   (network: (message(election(EID), ID_L, ID) NW)) S} =>
5  {(proc[ID]: process(initiator, LID, ID_L, ID_R))
6   (rcvLeft[ID]: election(EID)) (network: NW) S} .
```

ソースコード 5.3 は、書き換え規則 initiator-rcv-left の定義である。この書き換え規則は、initiator 状態のプロセス ID が、左側のプロセス ID_L から Election メッセージ message(election(EID), ID_L, ID) を受信した場合、rcvLeft にそのメッセージ種別を保持することを表している。

ソースコード 5.4: initiator-rcv-right の定義

```
1 rl [initiator-rcv-right] :
```

```
2  {(proc[ID]: process(initiator, LID, ID_L, ID_R))
3   (rcvRight[ID]: none)
4   (network: (message(election(EID), ID_R, ID) NW)) S} =>
5  {(proc[ID]: process(initiator, LID, ID_L, ID_R))
6   (rcvRight[ID]: election(EID)) (network: NW) S} .
```

ソースコード 5.4 は、書き換え規則 initiator-rcv-right の定義である。この書き換え規則は、initiator 状態のプロセス ID が、右側のプロセス ID_R から Election メッセージ message(election(EID), ID_R, ID) を受信した場合、rcvRight にそのメッセージ種別を保持することを表している。

ソースコード 5.5: initiator-become-leader の定義

```
1 crl [initiator-become-leader] :
2  {(proc[ID]: process(initiator, LID, ID_L, ID_R))
3   (rcvLeft[ID]: election(ID0)) (rcvRight[ID]: election(ID1))
4   (network: NW) S} =>
5  {(proc[ID]: process(leader, ID, ID_L, ID_R))
6   (rcvLeft[ID]: none) (rcvRight[ID]: none)
7   (network: (message(elected(ID), ID, ID_R) NW)) S}
8  if (ID0 >= ID1 and ID == ID0) or
9   (ID0 < ID1 and ID == ID1) .
```

ソースコード 5.5 は、書き換え規則 initiator-become-leader の定義である。この書き換え規則は、initiator 状態のプロセス ID が、両隣のプロセス ID_L および ID_R から受け取った Election メッセージに含まれるプロセス ID のうち、大きな方のプロセス ID と自身のプロセス ID を比較して等しい場合、プロセス ID は、leader 状態になることを表している。また、leader 状態になったプロセス ID は、右隣のプロセス ID_R に Elected メッセージ message(elected(ID), ID, ID_R) を送信する。

ソースコード 5.6: initiator-become-passive の定義

```
1 crl [initiator-become-passive] :
2  {(proc[ID]: process(initiator, LID, ID_L, ID_R))
3   (rcvLeft[ID]: election(ID0)) (rcvRight[ID]: election(ID1))
4   (network: NW) S} =>
5  {(proc[ID]: process(passive, LID, ID_L, ID_R))
6   (rcvLeft[ID]: none) (rcvRight[ID]: none) (network: NW) S}
7  if (ID0 >= ID1 and ID < ID0) or
8   (ID0 < ID1 and ID < ID1) .
```

ソースコード 5.6 は、書き換え規則 initiator-become-passive の定義である。この書き換え規則は、initiator 状態のプロセス ID が、両隣のプロセス ID_L および ID_R から受け取った Election メッセージに含まれるプロセス ID のうち、大きな方のプロセス ID と自身のプロセス ID を比較して小さい場合、プロセス ID は、passive 状態になることを表している。

ソースコード 5.7: initiator-repeat-election の定義

```
crl [initiator-repeat-election] :
 {(proc[ID]: process(initiator, LID, ID_L, ID_R))
  (rcvLeft[ID]: election(ID0)) (rcvRight[ID]: election(ID1))
  (network: NW) S} =>
 {(proc[ID]: process(initiator, LID, ID_L, ID_R))
  (rcvLeft[ID]: none) (rcvRight[ID]: none)
  (network: (
    message(election(ID), ID, ID_L)
    message(election(ID), ID, ID_R) NW)) S}
 if (ID0 >= ID1 and ID > ID0) or
  (ID0 < ID1 and ID > ID1) .
```

ソースコード 5.7 は、書き換え規則 initiator-repeat-election の定義である。この書き換え規則は、initiator 状態のプロセス ID が、両隣のプロセス ID_L および ID_R から受け取った Election メッセージに含まれるプロセス ID のうち、大きな方のプロセス ID と自身のプロセス ID を比較して大きい場合、プロセス ID は、両隣のプロセス ID_L および ID_R に Election メッセージ message(election(ID), ID, ID_L) と message(election(ID), ID, ID_R) を再送信することを表している。

ソースコード 5.8: normal-rcv-left の定義

```
rl [normal-rcv-left] :
 {(proc[ID]: process(normal, LID, ID_L, ID_R))
  (network: (message(election(EID), ID_L, ID) NW)) S} =>
 {(proc[ID]: process(passive, LID, ID_L, ID_R))
  (network: (message(election(EID), ID, ID_R) NW)) S} .
```

ソースコード 5.8 は、書き換え規則 normal-rcv-left の定義である。この書き換え規則は、normal 状態のプロセス ID が、左側のプロセス ID_L から Election メッセージ message(election(EID), ID_L, ID) を受信した場合、passive 状態になることを表している。また、受信した Election メッセージは、そのまま右側のプロセス ID_R に message(election(EID), ID, ID_R) として送信する。

ソースコード 5.9: normal-rcv-right の定義

```
rl [normal-rcv-right] :
 {(proc[ID]: process(normal, LID, ID_L, ID_R))
  (network: (message(election(EID), ID_R, ID) NW)) S} =>
 {(proc[ID]: process(passive, LID, ID_L, ID_R))
  (network: (message(election(EID), ID, ID_L) NW)) S} .
```

ソースコード 5.9 は、書き換え規則 normal-rcv-right の定義である。この書き換え規則は、normal 状態のプロセス ID が、右側のプロセス ID_R から Election メッセージ message(election(EID), ID_R, ID) を受信した場合、passive 状態になることを表している。また、受信した Election メッセージは、そのまま左側のプロセス ID_L に message(election(EID), ID, ID_L) として送信する。

ソースコード 5.10: passive-rcv-left の定義

```
1 rl [passive-rcv-left] :
2  {(proc[ID]: process(passive, LID, ID_L, ID_R))
3   (network: (message(election(EID), ID_L, ID) NW)) S} =>
4  {(proc[ID]: process(passive, LID, ID_L, ID_R))
5   (network: (message(election(EID), ID, ID_R) NW)) S} .
```

ソースコード 5.10 は、書き換え規則 passive-rcv-left の定義である。この書き換え規則は、passive 状態のプロセス ID が、左側のプロセス ID_L から Election メッセージ message(election(EID), ID_L, ID) を受信した場合、そのまま右側のプロセス ID_R に message(election(EID), ID, ID_R) として送信することを表している。

ソースコード 5.11: passive-rcv-right の定義

```
1 rl [passive-rcv-right] :
2  {(proc[ID]: process(passive, LID, ID_L, ID_R))
3   (network: (message(election(EID), ID_R, ID) NW)) S} =>
4  {(proc[ID]: process(passive, LID, ID_L, ID_R))
5   (network: (message(election(EID), ID, ID_L) NW)) S} .
```

ソースコード 5.11 は、書き換え規則 passive-rcv-right の定義である。この書き換え規則は、passive 状態のプロセス ID が、右側のプロセス ID_R から Election メッセージ message(election(EID), ID_R, ID) を受信した場合、そのまま左側のプロセス ID_L に message(election(EID), ID, ID_L) として送信することを表している。

ソースコード 5.12: passive-execution の定義

```
1 rl [passive-execution] :
2  {(proc[ID]: process(passive, LID, ID_L, ID_R))
3   (network: (message(elected(EID), ID_L, ID) NW)) S} =>
4  {(proc[ID]: process(passive, EID, ID_L, ID_R))
5   (network: (message(elected(EID), ID, ID_R) NW)) S} .
```

ソースコード 5.12 は、書き換え規則 passive-execution の定義である。この書き換え規則は、passive 状態のプロセス ID が、左側のプロセス ID_L から Elected メッセージ message(elected(EID), ID_L, ID) を受信した場合、プロセス ID が、リーダと認識しているプロセス ID を EID に変更することを表している。また、プロセス ID は、受信した Elected メッセージをそのまま右側のプロセス ID_R に message(elected(EID), ID, ID_R) として送信している。

ソースコード 5.13: leader-execution の定義

```
1 rl [leader-execution] :
2  {(proc[ID]: process(leader, LID, ID_L, ID_R))
3   (network: (message(elected(EID), ID_L, ID) NW)) S} =>
4  {(proc[ID]: process(leader, LID, ID_L, ID_R))
5   (network: NW) S} .
```

ソースコード 5.13 は、書き換え規則 leader-execution の定義である。この書き換え規則は、leader 状態のプロセス ID が、左側のプロセス ID_L から Elected メッセージ message(elected(EID), ID_L, ID) を受信した場合、その Elected メッセージを破棄することを表している。Franklin アルゴリズムでは、プロセスはリング状に配置されている。そのため、leader 状態のプロセスが、右側のプロセスに送信した Elected メッセージを、左側のプロセスから受信したことは、Elected メッセージがリングを一周したことを示唆している。したがって、Elected メッセージによる他のすべてのプロセスへの通知は終了したものとし、Elected メッセージを破棄する。

5.2.3 LTL 式

Franklin アルゴリズムが、活性を満たすことのモデル検査を行うために、以降に示す LTL 式を利用する。

ソースコード 5.14: FRANKLIN-PREDS モジュールの定義

```
1  mod FRANKLIN-PREDS is
2   pr FRANKLIN .
3   inc SATISFACTION .
4   subsort FState < State .
5   ops leader : Nat -> Prop .
6   vars ID LID ID_L ID_R : Nat .
7   var S : Soup{OComp} .
8   var PROP : Prop .
9   eq {(proc[ID]: process(leader, LID, ID_L, ID_R)) S}
10   |= leader(ID) = true .
11  eq {S} |= PROP = false [owise] .
12 endm
```

ソースコード 5.14 は、FRANKLIN-PREDS モジュールの定義である。このモジュールは、Franklin アルゴリズムの性質を LTL 式で表現するために必要な原子命題を定義している。

FRANKLIN-PREDS は、FRANKLIN および SATISFACTION をインポートしている。FRANKLIN は、5.2.2 項で説明した書き換え規則を定義したモジュールである。また、SATISFACTION は、Maude で LTL モデル検査器を利用するために必要な model-checker.maude に定義されているモジュールで、状態と原子命題の充足関係を表す |= 演算子が記述されている。FRANKLIN-PREDS では、|=演算子を利用して、LTL 式に必要な状態と原子命題の充足関係を定義している。FRANKLIN-PREDS に定義されている原子命題 leader(ID) は、状態が{(proc[ID]: process(leader, LID, ID_L, ID_R)) S}の場合に充足する。

ソースコード 5.15: FRANKLIN-CHECK モジュールの定義

```
1  mod FRANKLIN-CHECK is
```

```
2  inc FRANKLIN-PREDS .
3  inc MODEL-CHECKER .
4  inc LTL-SIMPLIFIER .
5  op leaderLiveness : -> Formula .
6  eq leaderLiveness =
7   <> (leader(0) \/ leader(1) \/ leader(2) \/
8    leader(3) \/ leader(4)) .
9 endm
```

ソースコード 5.15 は、FRANKLIN-CHECK モジュールの定義である。このモジュールは、モデル検査用のモジュールで Franklin アルゴリズムの性質を表す LTL 式を定義している。

FRANKLIN-CHECK では、3 つのモジュールをインポートしている。MODEL-CHECKER および LTL-SIMPLIFIER は、model-checker.maude に定義されているモジュールである。MODEL-CHECKER には、LTL モデル検査で利用する modelCheck 関数が記述されている。LTL-SIMPLIFIER には、LTL モデル検査のための補助モジュールで、LTL 式を簡略化する等式が定義されている。

FRANKLIN-CHECK では、LTL 式 leaderLiveness を定義している。leaderLiveness は、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質を表す LTL 式である。

5.3 Franklin アルゴリズムのモデル検査

本節では、前節で説明した形式仕様に基づいて、Franklin アルゴリズムが、リーダ選出アルゴリズムとしての正当性を満たしているかの確認をモデル検査で検証する。

リーダ選出アルゴリズムの正当性では次の 2 つの性質を検証する。

- 「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質
- 「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質

モデル検査を実施するにあたり、プロセスの並び順を考慮した初期状態を 2 つ定義する。1 つ目は、プロセスがプロセス ID の昇順にリング状に配置されている初期状態 init である。2 つ目は、プロセスがランダムにリング状に配置されている初期状態 init-random である。Franklin アルゴリズムの対象になるプロセスが 5 つで、それぞれのプロセス ID が 0 から 4 の場合、初期状態 init および初期状態 init-random を下記のように定義する。

ソースコード 5.16: 初期状態 init の定義

```
1 op {_} : Soup{OComp} -> FState [ctor] .
```

```
op init : -> FState .
eq init = {
 (proc[0]: process(normal, 0, 4, 1))
 (proc[1]: process(normal, 1, 0, 2))
 (proc[2]: process(normal, 2, 1, 3))
 (proc[3]: process(normal, 3, 2, 4))
 (proc[4]: process(normal, 4, 3, 0))
 (rcvLeft[0]: none) (rcvLeft[1]: none) (rcvLeft[2]: none)
 (rcvLeft[3]: none) (rcvLeft[4]: none)
 (rcvRight[0]: none) (rcvRight[1]: none) (rcvRight[2]: none)
 (rcvRight[3]: none) (rcvRight[4]: none)
 (network: empty)
} .
```

ソースコード 5.17: 初期状態 init-random の定義

```
op init-random : -> FState .
eq init-random = {
 (proc[0]: process(normal, 0, 2, 3))
 (proc[1]: process(normal, 1, 3, 4))
 (proc[2]: process(normal, 2, 4, 0))
 (proc[3]: process(normal, 3, 0, 1))
 (proc[4]: process(normal, 4, 1, 2))
 (rcvLeft[0]: none) (rcvLeft[1]: none) (rcvLeft[2]: none)
 (rcvLeft[3]: none) (rcvLeft[4]: none)
 (rcvRight[0]: none) (rcvRight[1]: none) (rcvRight[2]: none)
 (rcvRight[3]: none) (rcvRight[4]: none)
 (network: empty)
} .
```

ソースコード 5.16 は初期状態 `init`、ソースコード 5.17 は初期状態 `init-random` の定義である。各プロセスは、初期状態 init ではプロセス ID の昇順に、初期状態 `init-random` では 3、1、4、2、0 の順に右回りで配置されている。プロセスの状態は、両初期状態ともすべてのプロセスを `normal` 状態とし、リーダと認識しているプロセスは自身のプロセスとする。選任で左側および右側に配置されているプロセスから受け取ったメッセージ種別は、`none`（なし）とする。ネットワークワークは `empty` とし、ネットワーク上にメッセージはないものとする。

Franklin アルゴリズムのリーダ選出アルゴリズムとしての正当性を検証する。「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質は、ソースコード 4.15 のコマンドを、「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質は、ソースコード 4.16 のコマンドをそれぞれ実行することで確認できる。

ソースコード 5.18: 正当性における安全性の性質を確認するコマンド

```
search [1] in FRANKLIN : init =>*
 {(proc[ID0]: process(leader, LID0, ID0_L, ID0_R))
```

```
3   (proc[ID1]: process(leader, LID1, ID1_L, ID1_R)) S} .
4 search [1] in FRANKLIN : init-random =>*
5  {(proc[ID0]: process(leader, LID0, ID0_L, ID0_R))
6   (proc[ID1]: process(leader, LID1, ID1_L, ID1_R)) S} .
```

ソースコード 5.19: 正当性における活性の性質を確認するコマンド

```
1 red in FRANKLIN-CHECK : modelCheck(init, leaderLiveness) .
2 red in FRANKLIN-CHECK : modelCheck(init-random,
    leaderLiveness) .
```

ソースコード 5.18 およびソースコード 5.19 のコマンドを実行して、Maude から得られる結果は以下の通りである。

```
search [1] in FRANKLIN : init =>* {S (proc[ID0]: process(
   leader, LID0, ID0_L, ID0_R)) proc[ID1]: process(leader,
   LID1, ID1_L, ID1_R)} .

No solution.
states: 18494  rewrites: 389673 in 729ms cpu (730ms real)
   (533827 rewrites/second)
==========================================
search [1] in FRANKLIN : init-random =>* {S (proc[ID0]:
   process(leader, LID0, ID0_L, ID0_R)) proc[ID1]: process(
   leader, LID1, ID1_L, ID1_R)} .

No solution.
states: 21699  rewrites: 432689 in 818ms cpu (818ms real)
   (528939 rewrites/second)
==========================================
reduce in FRANKLIN-CHECK : modelCheck(init, leaderLiveness)
   .
rewrites: 482057 in 702ms cpu (702ms real) (686492 rewrites/
   second)
result Bool: true
==========================================
reduce in FRANKLIN-CHECK : modelCheck(init-random,
   leaderLiveness) .
rewrites: 541098 in 858ms cpu (866ms real) (630163 rewrites/
   second)
result Bool: true
```

ソースコード 5.18 のコマンドの実行結果は、初期状態 init および初期状態 init-random から、2 つの異なるプロセスが同時にリーダになる状態への遷移は見つからなかったことを示している。これにより、「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質が満たされることが分かった。次に、ソースコード 5.19 のコマンドの実行結果は、反例は検出されず、モデル検査が成功している。これにより、「いつか少なくとも 1 つのプロセスが、リー

ダに選ばれる」という活性の性質が満たされることが分かった。これらの結果から、Franklin アルゴリズムのリーダ選出アルゴリズムとしての正当性の性質が、満たされていることが確認できた。

5.4 まとめ

本章では、Franklin アルゴリズムの形式仕様を作成し、モデル検査を行った。モデル検査の実験では、Franklin アルゴリズムの対象となるプロセスを 5 つとし、プロセスの並び順を考慮した 2 つの初期状態において、Franklin アルゴリズムのリーダ選出アルゴリズムとしての正当性が満たされることを検証した。

リーダ選出アルゴリズムとしての正当性は、「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質と「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質を検証し、2 つの性質が満たされることを確認した。これにより、Franklin アルゴリズムは、リーダ選出アルゴリズムとしての正当性を満たしていることを検証できた。

なお、実験では Franklin アルゴリズムの対象となるプロセスの数を 6 以上にしてモデル検査を行ってみたが、初期状態のプロセスが 8 以上になると、モデル検査に時間がかかり、結果を得ることができなかった。

第6章　おわりに

本章では、各リーダ選挙アルゴリズムの形式仕様の作成やモデル検査の実験を通して得られた結論を報告する。また、各リーダ選出アルゴリズムの実験により得られた課題も報告する。

6.1　まとめ

本研究課題では、ブリーアルゴリズム、Chang-Roberts アルゴリズム、Franklin アルゴリズムの 3 つのリーダ選出アルゴリズムについて Maude にて形式仕様を作成した。また、作成した形式仕様を基に、各アルゴリズムにおいて保証すべき性質が満たされているかの確認をモデル検査で検証した。アルゴリズムの対象となるプロセス数の制限や書き換え規則などの形式仕様の作成に工夫をしてはいるが、現実的な検証時間でモデル検査を実行できている。以降は、各リーダ選出アルゴリズムごとに詳細を説明する。

6.1.1　ブリーアルゴリズム

ブリーアルゴリズムの選任処理では、あるプロセスがリーダの停止を検出すると次の選任処理を開始する。選任を開始したプロセスは、自身の ID より大きな ID をもつすべてのプロセスにメッセージを送信する。メッセージを送ったプロセスから返信がない場合は、選任のプロセスがリーダになり、選任のプロセスより大きな ID をもつプロセスから返信があった場合は、選任を交代する。最終的には、返信するプロセスがなくなり、残った 1 つのプロセスがリーダになる。本稿では、この選任処理の形式仕様を作成し、満たすべき性質の確認をモデル検査で検証した。

本稿の実験では、初期状態でアルゴリズムの対象となるプロセス数を 5 つにして、モデル検査を行った。モデル検査では、次の性質が満たされることを検証した。

- 「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質
- 「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質

ブリーアルゴリズムは、すべてのプロセスが、同期的にタイミングを合わせて1ステップずつ処理を行うため、本来、非同期で処理を実施するMaudeではモデル検査を行うことが難しい。Maudeで同期システムを扱うことの困難さを示す報告として、同期システムの形式仕様を作成しモデル検査を試みたが、状態空間爆発のためできなかったという報告[15]もある。この報告では、時間とリソースに制約のある単純なビジネスプロセスを、ラウンドベースのモデルとして形式化し、モデル検査をおこなったが、各ラウンドで無視できない数の局所遷移と中間状態が発生するため、モデル検査が不可能であったとされている。しかしながら、本稿では作成した形式仕様の書き換え規則に、同期的に遷移を進めるための対応を3つ取り入れることで、ブリーアルゴリズムのモデル検査を行っている。また、「いつか少なくとも1つのプロセスが、リーダに選ばれる」という活性の性質の検証では、書き換え規則による遷移がループしてしまい、反例が検出されてしまう問題を解決するために、強公平性を仮定して検証を行っている。

なお、実験では、初期状態のプロセス数を6以上にしたモデル検査も実施したが、検証に時間がかかり、結果を得ることができなかった。

6.1.2 Chang-Robertsアルゴリズム

Chang-Robertsアルゴリズムの選任処理は、非候補者のプロセスが、リーダの不在を検知すると開始される。リーダの不在を検知したプロセスは、始動プロセスとして候補者となり、隣のプロセスに自身のプロセスIDを含むメッセージを送る。メッセージを受け取ったプロセスが非候補者の場合は、メッセージをそのまま隣のプロセスに中継し、メッセージを受け取ったプロセスが候補者の場合は、自身のプロセスIDと受信したメッセージのプロセスIDを比較する。自身のプロセスIDの方が大きい場合は、受け取ったメッセージを隣に送る。反対に、自身のプロセスIDの方が小さい場合は、受け取ったメッセージを破棄する。この処理を最小のプロセスIDを含むメッセージだけがリングを一周するまで行う。本稿では、この選任処理の形式仕様を作成し、満たすべき性質の確認をモデル検査で検証した。

本稿の実験では、アルゴリズムの対象となるプロセス数を5つにし、並び順を考慮した初期状態を2つ定義して、モデル検査を行った。モデル検査では、次の性質が満たされることを検証した。

- 「常に2つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質
- 「いつか少なくとも1つのプロセスが、リーダに選ばれる」という活性の性質
- 「リーダの候補者になれば、いずれリーダになる」という活性の性質
- 「リーダの候補者になれば、いずれ自身が送信したcandidateメッセージはリングを一周する」という活性の性質

- 「リーダに選出されれば、いずれ自身が送信した coordinator メッセージはリングを一周する」という活性の性質

実験により、Chang-Roberts アルゴリズムは、リング状に配置されたプロセスの並び順に関係なく、リーダ選出アルゴリズムとしての正当性の性質が満たされていることを確認できた。また、Chang-Roberts アルゴリズム固有の 3 つの性質についても満たされていることが確認できた。

なお、実験では、初期状態のプロセス数を 6 以上にしたモデル検査も実施したが、初期状態のプロセスが 8 以上になると、検証に時間がかかり、結果を得ることができなかった。

6.1.3 Franklin アルゴリズム

Franklin アルゴリズムでは、非イニシエーターのプロセスが、リーダの不在を検知すると選任処理を開始する。リーダの不在を検知したプロセスは、イニシエーターとなり、両隣のプロセスに自身のプロセス ID を含むメッセージを送る。メッセージを受け取ったプロセスが非イニシエーターの場合は、パッシブプロセスとしてメッセージをそのまま隣のプロセスに中継するが、メッセージを受け取ったプロセスがイニシエーターの場合は、両隣のプロセスから受信したメッセージのプロセス ID のうち、大きな方のプロセス ID と自身のプロセス ID を比較する。自身のプロセス ID の方が大きい場合は、再度、両隣のプロセスに自身のプロセス ID を含むメッセージを送る。自身のプロセス ID の方が小さい場合は、パッシブプロセスとなる。自身のプロセス ID と等しい場合はリーダとなる。本稿では、この選任処理の形式仕様を作成し、満たすべき性質の確認をモデル検査で検証した。

本稿の実験では、アルゴリズムの対象となるプロセス数を 5 つにし、並び順を考慮した初期状態を 2 つ定義して、モデル検査を行った。モデル検査では、次の性質が満たされることを検証した。

- 「常に 2 つ以上の異なるプロセスが、同時にリーダに選ばれない」という安全性の性質
- 「いつか少なくとも 1 つのプロセスが、リーダに選ばれる」という活性の性質

実験により、Franklin アルゴリズムは、リング状に配置されたプロセスの並び順に関係なく、リーダ選出アルゴリズムとしての正当性の性質が満たされていることを確認できた。

なお、実験では、初期状態のプロセス数を 6 以上にしたモデル検査も実施したが、初期状態のプロセスが 8 以上になると、検証に時間がかかり、結果を得ることができなかった。

6.2 今後の課題

本稿では、3つの異なるリーダ選出アルゴリズムについて形式仕様を作成し、満たすべき性質の確認をモデル検査で検証した。これにより、現実的な検証時間で実施できるリーダ選出アルゴリズムの形式仕様の作成やモデル検査の方法を示すことはできたが、本研究課題を進めるにあたり、幾つかの課題が見えてきた。

1つ目の課題は、モデル検査の検証時間の長さについてである。本稿の実験では、すべてのリーダ選出アルゴリズムで、初期状態のプロセス数を5つにしてモデル検査を実施している。これはプロセス数を6つ以上で実行すると、状態爆発になり、モデル検査の検証時間が大幅に増えるためである。ブリーアルゴリズムは6つ、Chang-Roberts アルゴリズムと Franklin アルゴリズムは8つのプロセスを初期状態に指定するとモデル検査の実行に時間がかかり、結果を得られなかった。特にブリーアルゴリズムについては、JAIST の計算機サーバーを利用して、1週間検証を行ったが、結果を得ることができなかった。この課題については、状態数を減らす対応が必要である。

2つ目の課題は、Chang-Roberts アルゴリズムと Franklin アルゴリズムにおけるプロセスの並び順の検証についてである。リーダ選挙アルゴリズムの評価尺度には、メッセージ計算量がある。この評価尺度は、アルゴリズムの実行中に交換されたメッセージの総数で評価を行うが、Chang-Roberts アルゴリズムと Franklin アルゴリズムにはメッセージの交換数が最も多くなる並び順がある。だが、本稿の実験ではその並び順で行っていない。本稿の実験結果では、リング上のプロセスの並び順により、状態数が変化していた。そのため、プロセスの並び順によってはモデル検査の検証時間が変化するかの確認は必要だと考える。

付 録A　ソースコード

本稿の実験で用いた Maude のコードは、https://github.com/tarugo07/leader-election に登録している。

謝辞

本課題研究に取り組むにあたり、主指導教員としてご指導くださった緒方和博教授に心より感謝申し上げます。課題研究のテーマ設定、研究の遂行、課題研究報告書の執筆に至るまで、数々の貴重なご助言をいただきました。また、平石邦彦教授、青木利晃教授、石井大輔准教授にもお礼申し上げます。中間審査では、示唆に富むご指摘をいただきました。

参考文献

[1] H.Garcia-Molina. Elections in a distributed computing system., IEEE Trans. Comput., val.C-31, pp.48-59 (1982)

[2] アンドリュー・S・タネンバウム, マールテン・ファン・スティーン. 分散システム: 原理とパラダイム，ピアソン・エデュケーション (2009)

[3] 谷口秀夫. 分散処理, オーム社 (2005)

[4] E.Chang and R.Roberts. An improved algorithm for decentralized extrema-finding in circular configurations of processes., Communications of the ACM, vol.22, no.5, pp.281-283 (1979)

[5] 亀田恒彦, 山下雅史. 分散アルゴリズム, 近代科学社 (1994)

[6] W.R.Franklin. On an improved algorithm for decentralized extrema finding in circular configurations of processors., Communications of the ACM, vol.25, pp.336-337 (1982)

[7] L.Lamport. The part-time parliament., ACM Transactions on Computer Systems, val.16, no.2, pp.133-169 (1998)

[8] L.Lamport. Paxos made simple., ACM Sigact News, val.32, no.4 pp.18-25 (2001)

[9] D.Ongaro and J.Ousterhout. In search of an understandable consensus algorithm., Proc. Annual Technical Conference, pp.305-320, (2014)

[10] M.Clavel, et al. All About Maude - A High-Performance Logical Framework: How to Specify, Program, and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, val.4350, Springer (2007)

[11] Maude Manual (Version 3.2.1). https://maude.lcc.uma.es/maude321-manual-html/maude-manual.html.

[12] Leader Election in Distributed Systems. https://aws.amazon.com/builders-library/leader-election-in-distributed-systems/.

[13] 池田信之, 今村紀子, 高田沙都子. 実用化に向けたモデル検査適用手法の開発, 東芝レビュー, vol.6, no.9, pp.40-43 (2007)

[14] 中野伸哉, 小島英春, 土屋達弘. モデル検査器を用いたコンセンサスアルゴリズムの検証, 第 48 回（平成 27 年度）日本大学生産工学部学術講演会, pp.257-258 (2015)

[15] K.Ogata, T.Chaimanonta and M.Zhang. Formal modeling and analysis of time-and resource-sensitive simple business processes, Journal of Information Security and Applications, vol.31, pp.23-40 (2016)

[16] K.Ogata. A divide & conquer approach to liveness model checking under fairness & anti-fairness assumptions, Front. Comput. Sci., Vol.13, No.1, pp.51-72 (2019)