# 課題研究報告書

# 論理式肥大に伴う活性のモデル検査の非効率化の改善に関する調査

小柳 伶史

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３年３月

## 概 要

この課題研究では、公平性を仮定した活性のモデル検査で遭遇する、論理式肥大に伴う非効率化を改善をテーマに、分割統治アプローチ（divide & conquer approach（以下、DCA と呼ぶ））の有効性を検証する。本稿では Qlock と TAS の 2 つの事例研究を通して、当該アプローチの適用前後で場合わけをし、公平性を仮定した無排斥性のモデル検査をして比較した。Qlock はアトミックな待ち行列を利用する相互排除プロトコルで、Dijkstra のバイナリセマフォを抽象化したものである。TAS は、test & set のアトミックな命令を利用する相互排除プロトコルである。無排斥性とは活性の一つであり、プロセスが相互排除的に共用資源を利用したい場合、いつかは必ず利用することができる、という性質のことである。本稿の実験に用いた Qlock と TAS は、公平性を仮定しない場合には無排斥性は成り立たない。本課題研究は、これらの事例研究等を通して、DCA の有効性を確認できたことと、いくつかの得られた知見や、将来の方向性について報告する。

モデル検査は、高い品質が要求されるシステムにおいて、それを保証するために用いられる技術の一つであり、計算機科学におけるもっとも重要な成果のひとつだ。学術界で盛んに研究されているのみならず、産業界（特にハードウェア産業）でも日常業務で利用されている。活性のモデル検査では、しばしば公平性を仮定する必要がある。公平性とは、公平なプロセスなどのスケジューリングを抽象化したものである。公平性を利用するための一つの方法として、システムの性質を記述する線形時相論理式（LTL 式）の含意の前件として埋め込んで検査する方法がある。この方法で公平性を仮定した活性のモデル検査をすると、実現が困難という問題がしばしば発生する。これは、肥大化した LTL 式の Büchi オートマトンへの変換に時間が掛りすぎるからである。

近年では、このような公平性を仮定する場合に発生する非効率性の問題を解決するべく、いくつかの方法が研究、提案されている。本稿の調査対象である DCA は、システムの性質を表す LTL 式を、全体として真となるような、より小さな LTL 式に分割して個別に検査することで、モデル検査の実現を試みる。この方法は、専用のモデル検査器を改善する手法ではないことが特徴の一つである。ゆえに、性能面では専用のモデル検査器にはかなわないものの、既存のモデル検査器に広く適用できるため汎用性がとても高く、その重要性は高いと言える。

この課題研究報告書は 6 章構成である。第 1 章では本稿の背景と目的を説明し、後続の章の構造について説明する。第 2 章では Maude、LTL 式、公平性の仮定など、以降を読み進めるために必要となる予備知識を準備する。また、公平性を仮定した活性のモデル検査に対する DCA についても説明する。第 3 章と第 4 章では、Qlock と TAS の 2 つの事例について公平性の仮定の適用前後でモデル検査して、この手法を有効性の観点から比較する。第 5 章では、公平性の仮定をネイティ

ブに扱うことができる Process Analysis Tool（PAT）と Maude Fair LTLR model cheker の 2 つのモデル検査器について調査報告する。これらに加え、その他の一般的なモデル検査器が、公平性の仮定をどのようにサポートしているかをまとめると同時に、DCA を適用した公平性を仮定する活性のモデル検査と比較する。第 6 章は本稿の結論を述べるとともに、DCA の将来の方向性について言及する。

**キーワード:** モデル検査、公平性、活性、Maude、分割統治アプローチ（DCA）

# 目 次

**第 1 章　はじめに** **1**
1.1 背景 . . . . . 1
1.2 目的 . . . . . 2
1.3 構成 . . . . . 3

**第 2 章　準備** **4**
2.1 Maude . . . . . 4
2.2 状態機械 . . . . . 8
2.3 クリプケ構造 . . . . . 11
2.4 LTL の構文と意味 . . . . . 13
2.5 ラベル付きクリプケ構造 . . . . . 16
2.6 SE-LTL の構文と意味 . . . . . 16
2.7 EES クリプケ構造 . . . . . 17
2.8 公平性の仮定 . . . . . 21
2.9 分割統治アプローチ（DCA） . . . . . 24
2.10 まとめ . . . . . 26

**第 3 章　Qlock 相互排除プロトコルに対する無排斥性のモデル検査** **28**
3.1 Qlock 相互排除プロトコル . . . . . 28
3.2 クリプケ構造によるモデル検査 . . . . . 30
3.3 EES クリプケ構造によるモデル検査 . . . . . 35
3.4 DCA を適用したモデル検査 . . . . . 40
3.5 まとめ . . . . . 45

**第 4 章　TAS 相互排除プロトコルに対する無排斥性のモデル検査** **47**
4.1 TAS 相互排除プロトコル . . . . . 47
4.2 EES クリプケ構造によるモデル検査 . . . . . 48
4.3 DCA を適用したモデル検査 . . . . . 53
4.4 まとめ . . . . . 56

**第 5 章　関連研究** **57**
5.1 PAT(Process Analysis Toolkit) . . . . . 57
5.2 The Maude Fair LTLR Model Checker . . . . . 58

5.3 まとめ . . . . . 59

**第 6 章 おわりに** **61**
6.1 まとめ . . . . . 61
6.2 今後の課題 . . . . . 61

**付 録 A ソースコード** **63**

# 図 目 次

2.1 待ち行列の例 . . . . . 5
2.2 状態機械の例 . . . . . 8
2.3 初期状態のスープの例 . . . . . 10
2.4 遷移の例 . . . . . 11
2.5 三都市間の移動の例の反例 . . . . . 15
2.6 ラベル付き遷移の例 . . . . . 19

3.1 Qlock の例 . . . . . 29
3.2 クリプケ構造による Qlock の状態の例 . . . . . 31
3.3 Qlock の DCA 適用前後の比較 . . . . . 45

4.1 TAS の例 . . . . . 48
4.2 TAS の DCA 適用前後の比較 . . . . . 56

5.1 モデル検査器の公平性対応の一覧（2021 年 1 月 13 日時点） . . . . . 60

# 第1章　はじめに

本章では、この課題研究のテーマである、公平性を仮定した活性のモデル検査で遭遇する論理式肥大に伴う非効率化の改善と、その手法の一つである分割統治アプローチ（divide & conquer approach（以下、DCA と呼ぶ））について概説する。1.1 背景では、モデル検査が重要な技術である理由から始め、活性のモデル検査では公平性の仮定が必要となることと、その肥大した論理式が原因となる問題について説明する。1.2 目的では、この問題に対処する技術の一つである DCA の概説と調査方法、並びに本稿の達成目標を示す。1.3 構成で、全体の構成を示すことで本稿の導入とする。

## 1.1　背景

モデル検査（model checking）は、高い品質が要求されるシステムにおいて、それを保証するために用いられる技術の一つである。モデル検査は、計算機科学におけるもっとも重要な成果のひとつ [1] [1]で、学術界で盛んに研究されているのみならず産業界（特にハードウェア産業）でも日常業務で利用されている [2]。

モデル検査とは、検査したい対象のシステムを形式化したものに対し、そこで起こりうるあらゆる場合をしらみつぶしに調べることによって、そのシステムが期待通りに動作することを検証する技術のことである。システムの仕様は、状態機械の一種であるクリプケ構造として記述できる。システムが満たしてほしい性質は、線形時相論理式（LTL 式）として記述することができる。性質は、大きく安全性（safety）と活性（liveness）に分類でき、どちらも同等に重要である。安全性は、危険な状態には決して陥らないことを保証する性質である。活性は、目的が必ず達成できることを保証する性質である。

本稿で調査の対象としている、活性についてのモデル検査では、公平性（fairness）と呼ばれるものを仮定する必要がある。公平性とは、公平なプロセスなどのスケジューリングを抽象化したものである。公平性を仮定しない場合、活性のモデル検査結果は極端に偏った反例となって失敗する。公平性は、活性の一部、すなわち LTL 式の前件として記述することが可能である。しかしながら、Büchi オートマトンへの変換に時間が掛かりすぎるため、現実的なサイズの検査に対して適用

[1]Edmund Melson Clarke 、E. Allen Emerson 、Joseph Sifakis の 3 名は、モデル検査の発展に寄与したことでチューリング賞を受賞している。

することができない。近年では、このような公平性を仮定する場合の課題である「論理式肥大に伴う活性のモデル検査の非効率化」を解決するべく、いくつかの方法が研究、提案されている [3, 4, 5, 6, 7, 8, 9]。

## 1.2 目的

この課題研究は、公平性を仮定する活性のモデル検査を効率よく実現するための方法の一つである、2019 年に緒方が研究報告した DCA [3] の有効性を実証することを目的とする。DCA は、システムの性質を記述する LTL 式を分割することで小さくし、個別に統治（解決）することで効率化を図る。公平性を $fair$ で、活性を $\varphi$ であらわすと、検査対象の LTL 式は $fair \Rightarrow \varphi$ と記述できる。理論的にはこれを検査することで、モデル検査は成り立つ。しかし、LTL 式の Büchi オートマトンへの変換に、式の長さに応じて指数時間かかるため、現実的なサイズの検査に対処できない。DCA は、この肥大した論理式 $fair \Rightarrow \varphi$ を、以下の条件を満たす小さな論理式 $\varphi_1, \ldots, \varphi_n$ に分割し、それぞれモデル検査することで全体の正しさを確認する方法をとる。$fair \Rightarrow \varphi$ のモデル検査と $\varphi_1, \ldots, \varphi_n$ のモデル検査が等価であること、並びに各 $\varphi_i$ のモデル検査が可能であることが条件である。

有効性を確かめる方法として、本稿では実際に簡単な具体例を構築し、それに対して DCA の非適用/適用したモデル検査を実施し、効率性の観点から比較、考察する。本稿の実験に利用するモデル検査器として、[3] と同様に Maude モデル検査器を選択する。理由は、Maude は幾種もの並行・分散システムが記述された実績をもち、その表現力は豊かであり、処理速度もモデル検査に特化して開発された SPIN と同等であるという報告もある [10] ためだ。最後に、同一目的の別のアプローチを採用する近年の類似研究について調査することで、DCA の有効性をより深く理解する。

本課題研究の最終的な到達目標は、以上までに示したように、「論理式肥大に伴う活性のモデル検査の非効率化を改善するために提案された技術」である DCA に関して、一連の実験、調査を通してその有効性を実証し、それらの成果を調査結果として報告することである。なお、実験に用いた環境は以下の通りである。

- MacBook Air (Retina, 13-inch, 2018) パソコン
  - 1.6GHz デュアルコア Intel Core i5 プロセッサ
  - 8GB 2,133MHz LPDDR3 メモリ
- モデル検査器
  - Maude 2.7.1 LTL モデル検査器

## 1.3 構成

残りの章の構成は以下の通りである。

2. 本稿で必要となる用語や知識について、Maude モデル検査器を用いて説明する。
2.1 Maude の構文や命令について説明する。
2.2 システムの仕様を記述するために用いる状態機械を説明する。
2.3 状態機械を拡張した構造である、クリプケ構造を説明する。
2.4 システムの性質を記述できる LTL 式を、クリプケ構造において定める。
2.5 公平性の仮定を記述するために、ラベル付きクリプケ構造を説明する。
2.6 公平性を仮定できる SE-LTL 式を、ラベル付きクリプケ構造において定める。
2.7 ラベル付きクリプケ構造を模倣する、EES クリプケ構造を説明する。
2.8 EES クリプケ構造で可能となる、公平性を仮定したモデル検査を説明する。
2.9 分割統治アプローチ（DCA）を適用したモデル検査について説明する。
2.10 2 章を総括する。

3. Qlock を例として活性のモデル検査の実験例を再構築し、理解を深める。
3.1 Qlock 相互排除プロトコルについて説明する。
3.2 Qlock をクリプケ構造で記述し、モデル検査する。
3.3 Qlock を EES クリプケ構造で記述し、公平性を仮定してモデル検査する。
3.4 Qlock に DCA を適用し、効率化について観察する。
3.5 3 章を総括する。

4. TAS を例として活性のモデル検査を実験し、理解を深める。
4.1 TAS 相互排除プロトコルについて説明する。
4.2 TAS を EES クリプケ構造で記述し、公平性を仮定してモデル検査する。
4.3 TAS に DCA を適用し、効率化について観察する。
4.4 4 章を総括する。

5. 公平性の仮定をサポートしたモデル検査器の研究について、調査し報告する。
5.1 関連研究（PAT(Process Analysis Toolkit)）について報告する。
5.2 関連研究（The Maude Fair LTLR Model Checker）について報告する。
5.3 5 章を総括する。

6. 本研究課題を通して実験、考察したことを総括する。
6.1 DCA の有効性について、実験、観察した内容を元に報告する。
6.2 DCA の今後の課題について、改善提案を述べる。

# 第2章　準備

本章では、[3] で提示された「論理式肥大に伴う活性のモデル検査の非効率化を改善するために提案された技術」である分割統治アプローチ（DCA）を理解するために必要な、用語や知識を準備する。簡単な具体例を提示し、それを Maude のコードで形式化したものを用いて説明する。

## 2.1　Maude

本節では Maude について概説する。Maude の構文や命令などについて、具体的な例を Maude コードで記述し、簡単に説明する。

Maude は、C++で実装された言語であり、LTL モデル検査を標準機能としてサポートする。Maude では、実際に実行可能な形式仕様（formal executable specifications）としてアルゴリズムやシステム、言語、数学的構造などを記述することができる [11]。

待ち行列（queue）を具体例に、Maude の構文や命令などについて簡単に説明する。待ち行列は、スタックや木構造などと同じく、データ構造の一つである。待ち行列は、例えばシステムにおいて複数のプロセスが 1 つの資源を共用するときに必要とされる。待ち行列では、プロセスは次の動作を満たす。

- プロセスは、共用資源を使いたいとき、列の最後に並ぶ。
- プロセスは、列の先頭になるまで待機する。
- プロセスは、列の先頭になったら、共有資源を使うことを許可される。
- プロセスは、共用資源を使い終わったら列から離れる。

それでは、待ち行列の具体的な例として、人が行列をなして並ぶ例を考えてみよう（図 2.1）。最初の場面では、まだ列はない（空の列がある）。しだいに列ができ始め、alice、david、emma の順番に待ち行列ができる。次の場面では、alice は列の先頭を抜け、列には david、emma の 2 人が並んでいる。このとき、Maude では以下のように表すことができる。

```
1 fmod PID is
2   sort PersonID .
3   ops alice bob cathy david emma : -> PersonID [ctor] .
4 endfm
```

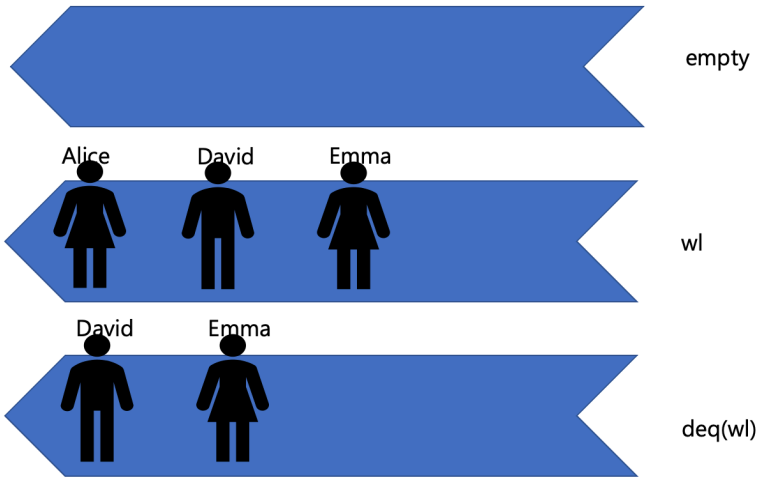


図 2.1: 待ち行列の例

```
5
6  fmod QUEUE is
7    pr PID .
8    sort Queue .
9    op empty : -> Queue [ctor] .
10   op _|_ : PersonID Queue -> Queue [ctor] .
11   op enq : Queue PersonID -> Queue .
12   op deq : Queue -> Queue .
13   var Q : Queue .
14   vars X Y : PersonID .
15   eq enq(empty,X) = X | empty .
16   eq enq((Y | Q),X) = Y | enq(Q,X) .
17   eq deq(empty) = empty .
18   eq deq((X | Q)) = Q .
19 endfm
20
21 fmod WAITINGLIST is
22   pr QUEUE .
23   op wl : -> Queue .
24   eq wl = enq(enq(enq(empty, alice), david), emma) .
25 endfm
26
27 red in WAITINGLIST : empty .
28 red in WAITINGLIST : wl .
29 red in WAITINGLIST : deq(wl) .
```

Listing 2.1: 待ち行列の Maude コード

モジュールは、**fmod**[1]キーワードを用いて定義でき、**endfm** までで一つとする。

[1]Maude システムに属する文字列は太字で示す。

ここでは3つのモジュール、PID、QUEUE、WAITINGLISTが定義されている。PIDは人識別子[2]を表している。QUEUEは待ち行列の構造、WAITINGLISSTは実際の列を記述している。プログラム末尾の27行目以降の行では、定義したモジュールWAITINGLISTを利用して待ち行列の動作を実験している。なお、命名規則は、見やすさ等を考慮して以下の通りとする。

- モジュール名は全て大文字である。
- ソート名は大文字から始まる。
- 演算子は小文字から始まる。

つづいて、それぞれのモジュールについて、該当する箇所のコードを抜粋して説明する。

モジュールPIDは、以下のように、ソートPersonIDと、その演算alice、bob、cathy、david、emmaとからなる。

```
1 fmod PID is
2   sort PersonID .
3   ops alice bob cathy david emma : -> PersonID [ctor] .
4 endfm
```

仕様記述において最初に必要となるのは、データ型の宣言と対応する演算の定義である。ソート（sort）とは、データの型のことである。代数仕様のコミュニティでは、型のことをソートと呼ぶことが多い[11]ため、本稿でもソートと呼ぶ。ソートは**sort**キーワードで宣言できる。ここでは、ソートPersonIDが定義されている。なお、Maudeでは文は**.**で終わる。[3]演算（operator）は、**op**キーワードを用いて宣言できる。3行目では、複数の演算が**ops**キーワードを用いて宣言されている。Maudeではキーワードの末尾が複数形になると、このように複数のトークンを記述することができる。演算の**:**の右側は、演算に関する定義を記述する。**->**の左側は、入力ソートの列、右側は出力ソートであり、こららの二つ組のことを演算のランクと呼ぶ。[と]とに囲まれた中に、属性を記述できる。**ctor**属性は、この演算が構成子（constructor）であることを意味する。他にも、重要な属性として、後述する**assoc**や**comm**などがある。例えば、3行目の演算aliceのランクは**->** PersonIDであり、0引数の演算、すなわち定数を示す。

モジュールQUEUEの前半では以下のように、ソートQueueと、演算empty、_|_、enq、deqとで、待ち行列の構造を定義している。

```
7   pr PID .
8   sort Queue .
```

[2]人識別子は、システムにおける擬人化したプロセスの識別子と考えてもらっても構わない。

[3]Maudeではトークンはスペースで区切られる。よくある構文ミスとして、**.**を入れ忘れる場合や、文末尾の**.**と直前のトークンとの間にスペースを入れない場合がある。

```
 9   op empty : -> Queue [ctor] .
10   op _|_ : PersonID Queue -> Queue [ctor] .
11   op enq : Queue PersonID -> Queue .
12   op deq : Queue -> Queue .
```

モジュールのはじめに、**pr** キーワードで先に定義したモジュール PID をインポートしている。empty と _|_ は構成子である。empty はそれだけで空の待ち行列を表し、_|_ は 2 つの引数をとることで空でない待ち行列を表す。**op** キーワードは、特殊な文字である _ を引数の位置を示す記号として用いる。 _ は、出現順に左から第一引数、第二引数、...、のように示す。ゆえに、_|_ は 2 つの引数をとる中置二項演算子（mixfix operator）であることを意味し、そのランクは PersonID Queue -> Queue である。enq は 2 引数の、deq は 1 引数の演算である。モジュール QUEUE の後半では、以下のように、演算 enq と deq に対して等式が定義されている。

```
13   var Q : Queue .
14   vars X Y : PersonID .
15   eq enq(empty,X) = X | empty .
16   eq enq((Y | Q),X) = Y | enq(Q,X) .
17   eq deq(empty) = empty .
18   eq deq((X | Q)) = Q .
```

変数は、**var** キーワードで宣言できる。変数はソート名を同時に宣言する。変数を複数同時に宣言するには **vars** キーワードを用いる。**op** キーワードで宣言した演算の等式は、**eq** キーワードで定義できる。ここでは、enq、deq は、それぞれ 2 行で 1 つの演算を定義している。enq は待ち行列の最後に並ぶときに利用される 2 引数の演算で、deq は待ち行列の先頭から離れるときに利用される 1 引数の演算である。

モジュール WAITINGLIST は、以下のように定義している。

```
21 fmod WAITINGLIST is
22   pr QUEUE .
23   op wl : -> Queue .
24   eq wl = enq(enq(enq(empty, alice), david), emma) .
25 endfm
```

先に定義したモジュール QUEUE をインポートし、演算 wl を定義することで、待ち行列の様子を記述している。ここでは、wl は空の待ち行列 empty に alice、david、emma の順番に並んだ (の順番に enq された) 待ち行列であることが示されている。

最後に、定義したモジュール WAITINGLIST を利用して、待ち行列の例を実験する。実験に用いる Maude コマンドは以下の通りである。

```
27 red in WAITINGLIST : empty .
28 red in WAITINGLIST : wl .
29 red in WAITINGLIST : deq(wl) .
```

実行（正確には、簡約（reduce））は **red**（または **reduce**）コマンドを用いる。ここでは、モジュール WAITINGLIST を指定して、WAITINGLIST 内に定義された演算 empty、wl、deq(wl) を実行している。これはすなわち、図 2.1 の

- 最初の場面では行列はまだなく（empty）、
- しだいに列ができ始め、alice、david、emma の順番に待ち行列ができ（wl）、
- alice は列の先頭を抜け、列には david、emma の 2 人が並んでいる（deq(wl)）

様子を表している。

## 2.2 状態機械

本節では、状態機械（state machine）について定義し、次に具体的な例を Maude コードで記述し説明する。

**定義 1 (状態機械)** 状態機械 $M$ を、$\langle S, I, T \rangle$ からなる三つ組と定める。$S$ は状態の集合である。$I$ は初期状態の集合で、$I \subseteq S$ である。$T$ は全域的（*total*）な二項関係で、$T \subseteq S \times S$ である。$(s, s') \in T$ は $s \to s'$ で表し、これを遷移（*transition*）と呼ぶ。

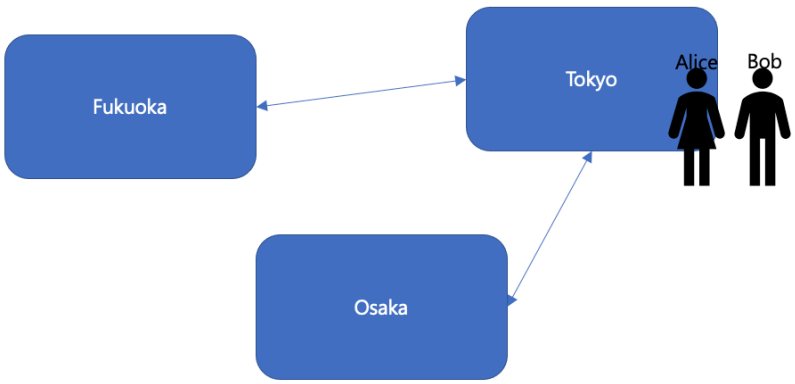


図 2.2: 状態機械の例

それでは、次のような例を考えてみよう。alice と bob は同僚で、tokyo の本店で働いている。支店が fukuoka と osaka とにある。2 人はめいめい、本店と支店の

ある三都市間を頻繁に移動している。ただし支店間の移動はなく、支店へ移動した場合は直接本店へ戻るものとする。

この例を状態機械 $M_{\mathrm{exm1}} \triangleq \langle S_{\mathrm{exm1}}, I_{\mathrm{exm1}}, T_{\mathrm{exm1}} \rangle$ で定め、Maude コードとして記述した形式仕様を提示し、それぞれ説明する。

状態の集合 $S_{\mathrm{exm1}}$ を Maude コードで記述したものは以下の通りである。

```
1  *** S_{exm1} BEGIN
2  fmod PID is
3    sort PersonID .
4    ops alice bob : -> PersonID [ctor] .
5  endfm
6
7  fmod LOCATION is
8    sort Location .
9    ops tokyo osaka fukuoka : -> Location [ctor] .
10 endfm
11
12 fmod SOUP is
13   pr PID .
14   pr LOCATION .
15   sorts OComp Soup .
16   subsort OComp < Soup .
17   *** observable component
18   op name[_]:_ : PersonID Location -> OComp [ctor] .
19   *** soup
20   op __ : Soup Soup -> Soup [ctor assoc comm] .
21 endfm
22 *** S_{exm1} END
```

Listing 2.2: $S_{\mathrm{exm1}}$

前節と同様、モジュール PID でソート PersonID とその定数 alice、bob を定める。モジュール LOCATION で、ソート Location の定数として tokyo、osaka、fukuoka を定める。つづいて、状態のデータ構造をモジュール SOUP として定める。SOUP は、先に定めた PID と LOCATION をインポートし、ソートとして OComp と Soup を宣言する。さらに **subsort** キーワードで OComp が Soup のサブソートであることを定める。ソートのサブソート関係 $<$ は、半順序である。

次に状態の表現方法について説明する。状態をデータ構造として表現する方法は一つではない。本稿では状態を、結合法則（assosciative law）ならびに交換法則（commutativite law）を満たす、名前と値の対（name-value pair）のコレクションとして表現する。Maude のオンラインマニュアル [11] に倣い、結合法則と交換法則を満たしたコレクションのことをスープ（soup）、名前と値の対のことを観測可能成分（observable component）と呼ぶ。つまり本稿では状態を、観測可能成分のスープとして表現する。18 行目で、観測可能成分を 2 項演算 name[_]:_として定める。第一引数に PersonID、第二引数に Location をとる構成子である。20 行目で、

観測可能成分のスープを __ すなわち juxtaposition 演算子[4]として定める。属性として **ctor** で構成子であることを、**assoc** と **comm** を指定し結合法則と交換法則とを満たすことを定める。OComp はサブソート関係で定める通り、それ単体で一つの Soup であるし、 __ のランク Soup Soup -> Soup が示す通り、複数の Soup も一つの Soup となる。

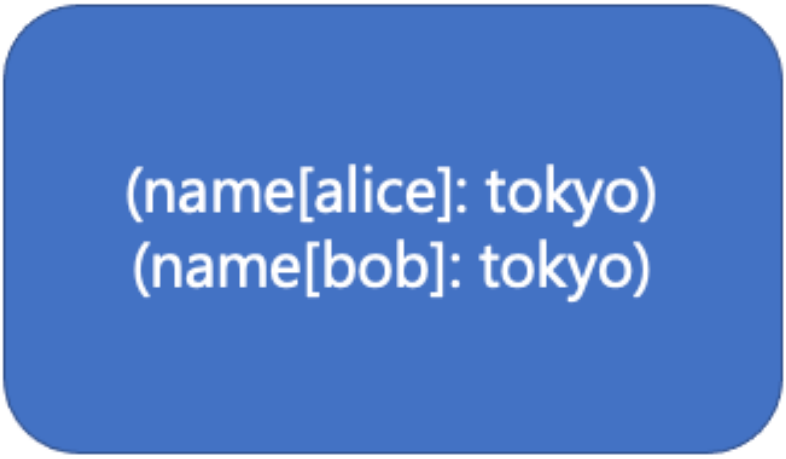


図 2.3: 初期状態のスープの例

初期状態の集合 $I_{\text{exm1}}$ を Maude コードで記述したものは以下の通りである。

```
*** I_{exm1}
fmod INIT-SOUP is
  pr SOUP .
  op init : -> Soup .
  eq init = (name[alice]: tokyo) (name[bob]: tokyo) .
endfm
```

Listing 2.3: $I_{\text{exm1}}$

初期状態 init は、alice、bob は tokyo に位置する、観測成分が 2 つのスープとして定める。

遷移の集合 $T_{\text{exm1}}$ は、以下のような書き換え規則で表現する。

```
*** T_{exm1}
mod TRANSITION is
  pr SOUP .
  var I : PersonID .
  rl [t2o] : (name[I]: tokyo)
```

[4]juxtaposition 演算子は、通常の演算子と違いその記号が陽には見えない演算子のことである。Maude では **op** キーワードの _ 記号のみで記述することにより juxtaposition 演算子を定義できる。

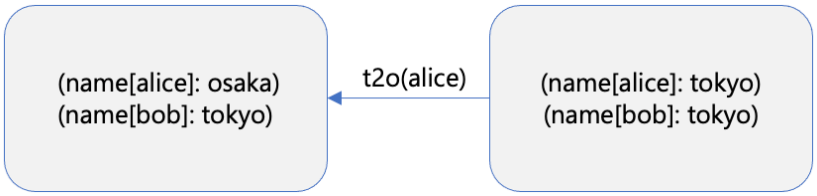


図 2.4: 遷移の例

```
36            => (name[I]: osaka) .
37   rl [o2t] : (name[I]: osaka)
38            => (name[I]: tokyo) .
39   rl [t2f] : (name[I]: tokyo)
40            => (name[I]: fukuoka) .
41   rl [f2t] : (name[I]: fukuoka)
42            => (name[I]: tokyo) .
43 endm
```

Listing 2.4: $T_{\mathrm{exm1}}$

遷移は、**rl** キーワードを用いて書き換え規則で表現する。ここでは 4 つの書き換え規則が定められている。なお、Maude では改行は構文的に意味を持たないため、読みやすさのため適宜挿入してある。例えば 35 行目の書き換え規則 [t2o] は、人識別子を表す変数 `I` が場所 `tokyo` から `osaka` に移る、`t2o`（`tokyo to osaka`）という名称の（複数の）遷移であることを表現している。

## 2.3 クリプケ構造

本節では、クリプケ構造（`Kripke structures`）について定義し、次に具体的な例を `Maude` コードで記述し説明する。クリプケ構造は、状態機械を拡張した構造である。`Maude` では、クリプケ構造を用いて検査対象のシステムを形式化することで、モデル検査することができる。

**定義 2（クリプケ構造）** クリプケ構造 $K$ を、次のような五つ組 $\langle S, I, P, L, T\rangle$ と定める。$S$ は状態の集合である。$I$ は初期状態の集合で、$I \subseteq S$ である。$P$ は原子状態命題（*`atomic state proposition`*）の集合で、$P \subseteq U$ である。$L$ はラベリング関数で、その型は $S \to 2^P$ である。$T$ は全域的（*`total`*）な二項関係で、$T \subseteq S \times S$ である。

それでは、先の状態機械の例と同様の例、`alice` と `bob` が働いている三都市間の移動の例をクリプケ構造 $K_{\mathrm{exm2}} \triangleq \langle S_{\mathrm{exm2}}, I_{\mathrm{exm2}}, P_{\mathrm{exm2}}, L_{\mathrm{exm2}}, T_{\mathrm{exm2}}\rangle$ で定め、`Maude` コードとして記述した形式仕様を提示する。前節 `2.2` との違いは、各人の、各都市に対応する述語を定め、この述語を用いてモデル検査を可能とすることにある。な

お、$S_{\text{exm2}}$、$I_{\text{exm2}}$、$T_{\text{exm2}}$ については、$S_{\text{exm2}} = S_{\text{exm1}}$、$I_{\text{exm2}} = I_{\text{exm1}}$、$T_{\text{exm2}} = T_{\text{exm1}}$ であるためコードは再掲しない。

原子状態命題の集合 $P_{\text{exm2}}$ とラベリング関数 $L_{\text{exm2}}$ を Maude コードで記述したものは以下の通りである。

```
in model-checker .
*** P_{exm2}, L_{exm2}
mod PREDICATES is
  pr TRANSITION .
  inc SATISFACTION .
  subsort Soup < State .
  *** P_{exm2}
  op inTokyo : PersonID -> Prop .
  op inOsaka : PersonID -> Prop .
  op inFukuoka : PersonID -> Prop .
  var I : PersonID .
  var S : Soup .
  var PR : Prop .
  *** L_{exm2}
  eq (name[I]: tokyo) S   |= inTokyo(I) = true .
  eq (name[I]: osaka) S   |= inOsaka(I) = true .
  eq (name[I]: fukuoka) S |= inFukuoka(I) = true .
  eq S |= PR = false [owise] .
endm
```

Listing 2.5: $P_{\text{exm2}}$ と $L_{\text{exm2}}$

ここでは、原子状態命題の集合 $P_{\text{exm2}}$ を演算、ラベリング関数 $L_{\text{exm2}}$ を充足関係として定義している。まず、先に定義したモジュール TRANSITION と、Maude の LTL モデル検査用の model-checker.maude ファイル[5]に定義されているモジュール SATISFACTION をインポートしている。SATISFACTION は、状態と原子状態命題に対して真偽を定め、充足関係を表すことができる。ソート Soup を、SATISFACTION で定義されているソート State のサブソートとして定めることで、演算 _ |= _ を利用することができる。これは、ソート State と Prop をとってソート Bool を返す演算である。[6]原子状態命題はそれぞれ、PersonID をとって Prop を返す演算 inTokyo、inOsaka、inFukuoka として定めることで、$P_{\text{exm2}}$ = {inTokyo(alice), inOsaka(alice), inFukuoka(alice), inTokyo(bob), inOsaka(bob), inFukuoka(bob)} を表す。変数 I は人識別子、S は観測可能成分のスープ、PR は原子状態命題である。

ラベリング関数 $L_{\text{exm2}}$ は eq キーワードを用いて複数行の等式で（ここでは 4 行で）定義する。例えばスープ (name[alice] : tokyo)(name[bob] : fukuoka) を受け

[5] **in** コマンドでファイルを読みこむことができる。

[6] 例えば **red** (name[alice]: tokyo) (name[bob]: fukuoka) |= inFukuoka(bob) を実行すると true が返る。これは、「状態 (name[alice]: tokyo) (name[bob]: fukuoka) で原子状態命題 inFukuoka(bob) を充足する」のように読む。

取り、原子状態命題の集合 $\{\mathrm{inTokyo(alice)}, \mathrm{inFukuoka(bob)}\}$ を返すような関数である。

次節では、モデル検査をしたい性質を記述するために用いられる、LTL 式の構文と意味について説明し、三都市間の移動の例に対し、実際にモデル検査を実施する。

## 2.4 LTL の構文と意味

本節では、クリプケ構造において LTL 式の構文と意味を定める。

任意のクリプケ構造 $K$ に対し、$K$ のパス（path）とは、自然数 $i \geq 0$ において、$s_i \rightarrow s_{i+1}$ なる無限列 $s_0; \ldots; s_i; s_{i+1}; \ldots$ である。$K$ の実行系列（computation）とは、$\pi(0) \in I$ なるパス $\pi$ のことである。$\mathcal{P}$ を $K$ の全てのパスの集合、$\mathcal{C}$ を $K$ の全ての実行系列の集合と仮定する。このとき、任意のクリプケ構造 $K$ に対し、$K$ の LTL 式 $\varphi$ の構文を次のように定める。

$\varphi ::= \ \top \mid p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \bigcirc\varphi \mid \varphi \ \mathcal{U} \ \varphi$

ここで $p$ は、$p \in P$ である。$\mathcal{F}$ を $K$ の全ての LTL 式の集合と仮定する。

**定義 3（LTL の意味）** 任意のクリプケ構造 $K$ と、$K$ の任意のパス $\pi \in \mathcal{P}$ と、$K$ の任意の *LTL* 式 $\varphi \in \mathcal{F}$ に対して、$K, \pi \models \varphi$ を以下のように帰納的に定める。

- $\varphi$ が $\top$ のとき、$K, \pi \models \top$ である。
- $\varphi$ が $p \in P$ の場合、さらに $K, \pi \models p$ のとき、かつそのときに限り、$p \in L(\pi(0))$ である。
- $\varphi$ が $\neg\varphi_1$ のとき、さらに $K, \pi \models \neg\varphi_1$ のとき、かつそのときに限り、$K, \pi \not\models \varphi_1$ である。
- $\varphi$ が $\varphi_1 \wedge \varphi_2$ のとき、さらに $K, \pi \models \varphi_1 \wedge \varphi_2$ のとき、かつそのときに限り、$K, \pi \models \varphi_1$ かつ $K, \pi \models \varphi_2$ である。
- $\varphi$ が $\bigcirc\varphi_1$ のとき、さらに $K, \pi \models \bigcirc\varphi_1$ のとき、かつそのときに限り、$K, \pi^1 \models \varphi_1$ である。
- $\varphi$ が $\varphi_1 \ \mathcal{U} \ \varphi_2$ のとき、さらに $K, \pi \models \varphi_1 \ \mathcal{U} \ \varphi_2$ のとき、かつそのときに限り、ある自然数 $i$ が存在して $K, \pi^i \models \varphi_2$ かつ全ての自然数 $j < i$ に対して $K, \pi^j \models \varphi_1$ である。

ここで、$\varphi_1$ と $\varphi_2$ は LTL 式である。これらにより、$K \models \varphi$ は、任意の $\pi \in \mathcal{C}$ に対し $K, \pi \models \varphi$ と定まる。

時相演算子（temporal connectives）$\bigcirc$ と $\mathcal{U}$ はそれぞれ、next 演算子と until 演算子と呼ばれる。そのほかの時相演算子は以下のように定義する。

- $\bot \triangleq \neg\top$
- $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$
- $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$
- $\Diamond\varphi \triangleq \top\ \mathcal{U}\ \varphi$
- $\Box\varphi \triangleq \neg(\Diamond\neg\varphi)$
- $\varphi_1 \rightsquigarrow \varphi_2 \triangleq \Box(\varphi_1 \Rightarrow \Diamond\varphi_2)$

時相演算子 $\Diamond$、$\Box$、$\rightsquigarrow$ はそれぞれ、eventually 演算子、always 演算子、leadsto 演算子と呼ばれる。

モデル検査用に以下のように定めたモジュール 3CITIES-CHECK について説明する。

```
65 mod 3CITIES-CHECK is
66   pr INIT-SOUP .
67   inc PREDICATES .
68   inc MODEL-CHECKER .
69   inc LTL-SIMPLIFIER .
70   vars P1 P2 : PersonID .
71   var S : Soup .
72   op liveness : -> Formula .
73   eq liveness = (<> inFukuoka(alice)) /\ (<> inFukuoka(bob))
         .
74 endm
```

Listing 2.6: 三都市間の移動のモデル検査

モデル検査用モジュール 3CITIES-CHECK は、以下の 4 つのモジュールをインポートしている。INIT-SOUP は $I_{\mathrm{exm2}}$ で、2 人が tokyo にいる状態を示し、PREDICATES は先に定めた $P_{\mathrm{exm2}}$ と $L_{\mathrm{exm2}}$ である。MODEL-CHECKER [7]は LTL モデル検査に必須のモジュールで、modelCheck() 関数が定義されている。 LTL-SIMPLIFIER [8]はモデル検査の補助モジュールである。変数 P1、P2 と S は、modelCheck() の実行時に利用するためにここで宣言する。liveness のソートは Formula [9]で、これはモデル検査をしたい性質の LTL 式を意味する。システムが満たすべき性質は安全性（safety）と活性（liveness）に大別されるが、LTL 式 liveness は活性に分類される。安全性は、「悪いことが起こらないことを保証する性質」のことである。活性は、「良いことが、将来いつかは必ず起こることを保証する性質」のことであ

[7]MODEL-CHECKER は、先に読み込んだファイル model-checker.maude に定義されている。
[8]LTL-SIMPLIFIER は、先に読み込んだファイル model-checker.maude に定義されている。
[9]ソート Formula とその演算は、先に読み込んだ model-checker.maude でモジュール LTL に定義されている。

る。73 行目は、$(\Diamond \text{inFukuoka(alice)}) \wedge (\Diamond \text{inFukuoka(bob)})$ という LTL 式、すなわち「いつかは inFukuoka(alice) かつ、いつかは inFukuoka(bob)」と読む。これは、alice はいつかは福岡へ行くことと、 bob はいつかは福岡へ行くこととが成り立つことを表現しており、これは活性である。

以下のコードを実行することで、実際にモデル検査をする。

```
75 red in 3CITIES-CHECK : modelCheck(init, liveness) .
```

Listing 2.7: 三都市間の移動のモデル検査の実行

このとき、以下の反例が返ってきてモデル検査は失敗する。

```
reduce in 3CITIES-CHECK : modelCheck(init, liveness) .
rewrites: 44 in 0ms cpu (0ms real) (97995 rewrites/second)
result ModelCheckResult: counterexample(
    {(name[alice]: tokyo) name[bob]: tokyo,'t2o}
    {(name[alice]: osaka) name[bob]: tokyo,'t2o},
    {(name[alice]: osaka) name[bob]: osaka,'o2t}
    {(name[alice]: tokyo) name[bob]: osaka,'t2o})
```

Listing 2.8: 三都市間の移動のモデル検査の実行結果

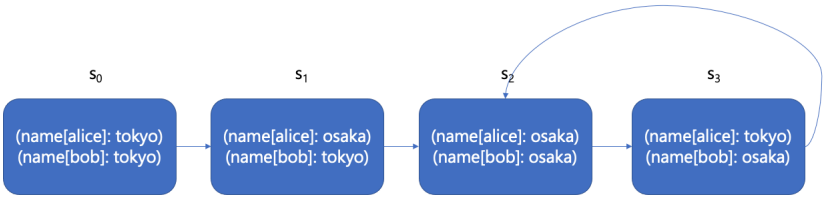


図 2.5: 三都市間の移動の例の反例

今回は、反例（counterexample）が検出されてしまいモデル検査は失敗する。出力結果が反例の場合は、自然数 $i < n$ のとき、全体で 1 つの実行系列 $\pi = s_0; \ldots; s_i; (s_{i+1}; \ldots; s_n)^\infty$ を表す無限列を返す。ここで列は、状態と遷移名とからなる状態遷移の二つ組として表現されている。さらに、出力結果の counterexample() は列と列の二つ組として表現されていることに注意して欲しい。1 番目の列は $\pi$ の $s_0; \ldots; s_i$ に、2 番目の列は $\pi$ の $(s_{i+1}; \ldots; s_n)^\infty$ に該当する。

したがって、今回の反例は、1 番目の状態は初期状態で、遷移名は't2o から始まり、2 番目の状態は alice が osaka、bob が tokyo で、遷移名't2o となる。次の 3 番目と 4 番目はループする列を示す。3 番目の状態が alice が osaka、bob が osaka で、遷移名'o2t。4 番目の状態が alice が tokyo、bob が osaka で、遷移名't2o であるため、alice も bob も fukuoka へ位置することなく、alice は永遠に tokyo と osaka の間を無限ループして移動し続けるという反例が検出される。

このような極端に偏った失敗例は、モデル検査の反例としては望ましいものではない。そこで、モデル検査をしたい性質に対し、公平性を仮定した上で検査を行うという手法を用いる。しかしながら、公平性の仮定は通常のクリプケ構造では表現できず、また、Maude は通常のクリプケ構造しか記述することができない。

この問題を解決するためにはどうすれば良いか。まず、公平性の仮定の記述には、クリプケ構造を拡張した、ラベル付きクリプケ構造を用いる必要がある。そして、このラベル付きクリプケ構造は、通常のクリプケ構造で模倣することができる。ゆえに、公平性を仮定した Maude のモデル検査は、「ラベル付きクリプケ構造を模倣する（通常の）クリプケ構造」を用いて記述することによって可能となる。

## 2.5 ラベル付きクリプケ構造

本節では、ラベル付きクリプケ構造（Labeled Kripke structures（LKS））について説明する。ラベル付きクリプケ構造を用いるとモデル検査において公平性の仮定を記述できるようになる。ラベル付きクリプケ構造は、クリプケ構造に対して、イベントの集合を加えて定義される。イベントは、状態遷移の名前である。

**定義 4（ラベル付きクリプケ構造）** ラベル付きクリプケ構造 $lK$ は次のような六つ組 $\langle lS, lI, lE, lP, lL, lT\rangle$ と定める。$lS$ は状態の集合である。$lI$ は初期状態の集合で、$lI \subseteq lS$ である。$lE$ はイベントの集合で、$lE \subseteq U$ である。$lP$ は原子状態命題の集合で、$lP \subseteq U$ かつ $lP \cap lE = \emptyset$ である。$lL$ はラベリング関数で、その型は $lS \to 2^{lP}$ である。$lT$ は全域的な三項関係で、$lT \subseteq lS \times lE \times lS$ である。$(s, e, s') \in lT$ は $s \to_e s'$ で表し、ラベル付き遷移 $e$（または単純に、遷移 $e$）と呼ぶ。あるイベント $e \in lE$ が存在して、どんな $s, s' \in lS$ に対しても $(s, e, s') \notin lT$ であるような $e$ のことを $\iota$ と呼ぶ。

それぞれのイベント $e \in lE$ に対し、ある原子状態命題 $\mathrm{enabled}(e) \in lP$ が存在すると仮定する。それぞれの状態 $s \in lS$ に対し $\mathrm{enabled}(e) \in lL(s)$ のとき、かつそのときに限り、ある $s' \in lS$ が存在して $(s, e, s') \in lT$ と仮定する。ラベル付き遷移 $e$ が状態 $s$ で適用されるとき、かつそのときに限り、$\mathrm{enabled}(e) \in lL(s)$ となる。

任意のラベル付きクリプケ構造 $lK$ に対し、$lK$ のパスとは、自然数 $i \geq 0$ において、$(s_i, e_{i+1}, s_{i+1}) \in lT$ なる $lE \times lS$ の無限列 $(e_0, s_0); \ldots; (e_i, s_i); (e_{i+1}, s_{i+1}); \ldots$ である。$lK$ の実行系列とは、$1 \bullet l\pi(0) = \iota$ かつ $2 \bullet l\pi(0) \in lI$ となるパス $l\pi$ のことである。$l\mathcal{P}$ を $lK$ の全てのパスの集合、$l\mathcal{C}$ を $lK$ の全ての実行系列の集合と仮定する。

## 2.6 SE-LTL の構文と意味

本節では、ラベル付きクリプケ構造において SE-LTL 式（State/Event-based linear temporal logic formula）の構文と意味について定める。

任意のラベル付きクリプケ構造 $lK$ に対し、$lK$ の SE-LTL 式 $l\varphi$ の構文を次のように定める。

$l\varphi ::= \ \top \mid p \mid e \mid \neg l\varphi \mid l\varphi \wedge l\varphi \mid \bigcirc l\varphi \mid l\varphi \ \mathcal{U} \ l\varphi$

ここで $p$ は $p \in lP$、$e$ は $e \in lE$ である。$l\mathcal{F}$ を $lK$ の全ての LTL 式の集合と仮定する。

**定義 5（SE-LTL の意味）** 任意のラベル付きクリプケ構造 $lK$ と、$lK$ の任意のパス $l\pi \in l\mathcal{P}$ と、$lK$ の任意の *SE-LTL* 式 $l\varphi \in l\mathcal{F}$ に対して、$lK, l\pi \models l\varphi$ を以下のように帰納的に定める。

- $l\varphi$ が $\top$ のとき、$lK, l\pi \models \top$ である。
- $l\varphi$ が $p \in lP$ の場合、さらに $lK, l\pi \models p$ のとき、かつそのときに限り、$p \in lL(2 \bullet l\pi(0))$ である。
- $l\varphi$ が $e \in lE$ の場合、さらに $lK, l\pi \models e$ のとき、かつそのときに限り、$e = lL(1 \bullet l\pi(0))$ である。
- $l\varphi$ が $\neg l\varphi_1$ のとき、さらに $lK, l\pi \models \neg l\varphi_1$ のとき、かつそのときに限り、$lK, l\pi \not\models l\varphi_1$ である。
- $l\varphi$ が $l\varphi_1 \wedge l\varphi_2$ のとき、さらに $lK, l\pi \models l\varphi_1 \wedge l\varphi_2$ のとき、かつそのときに限り、$lK, l\pi \models l\varphi_1$ かつ $lK, l\pi \models l\varphi_2$ である。
- $l\varphi$ が $\bigcirc l\varphi_1$ のとき、さらに $lK, l\pi \models \bigcirc l\varphi_1$ のとき、かつそのときに限り、$lK, l\pi^1 \models l\varphi_1$ である。
- $l\varphi$ が $l\varphi_1 \ \mathcal{U} \ l\varphi_2$ のとき、さらに $lK, l\pi \models l\varphi_1 \ \mathcal{U} \ l\varphi_2$ のとき、かつそのときに限り、ある自然数 $i$ が存在して $lK, l\pi^i \models l\varphi_2$ かつ全ての自然数 $j < i$ に対して $lK, l\pi^j \models l\varphi_1$ である。

ここで、$l\varphi_1$ と $l\varphi_2$ は SE-LTL 式である。これらにより、$lK \models l\varphi$ は、任意の $l\pi \in l\mathcal{C}$ に対し $lK, l\pi \models l\varphi$ と定まる。

$lK, l\pi \models e$ とは、ラベル付き遷移 $e$ が状態 $2 \bullet l\pi(0)$ において、$lK$ に関して `applied(`$e$`)` となることを意味する。

## 2.7 EES クリプケ構造

Maude ではラベル付きクリプケ構造を記述できない。Maude で記述できるよう、ラベル付きクリプケ構造を模倣する（通常の）クリプケ構造について説明する。

ラベル付きクリプケ構造は、イベントをクリプケ構造に状態として埋め込むことで模倣することができる。これは「ラベル付きクリプケ構造の EES クリプケ構造

(Event-embedded-in-states Kripke Structures)」と呼ばれる。名称として長いため、以降は、LKS の EES-KS（または単に EES-KS）と呼称する。LKS の SE-LTL 式は、それを模倣する EES-KS の LTL 式で意味が保存され、逆も成り立つ [3] これにより、Maude LTL モデル検査器では直接扱えないラベル付きクリプケ構造と SE-LTL 式は、同等の意味を持つ EES-KS と LTL 式により扱うことができるようになり、公平性を仮定したモデル検査を行うことができる。

**定義 6（EES クリプケ構造）**任意のラベル付きクリプケ構造 $lK$ に対して、$lK$ の ***EES*** クリプケ構造 $K_{\mathrm{ees}}$ は次のような五つ組 $\langle S_{\mathrm{ees}}, I_{\mathrm{ees}}, P_{\mathrm{ees}}, L_{\mathrm{ees}}, T_{\mathrm{ees}}\rangle$ である。状態の集合 $S_{\mathrm{ees}} = \{(e,s)|e \in lE, s \in lS\}$ は、$lS \times lE$ である。初期状態の集合 $I_{\mathrm{ees}}$ は $\{(\iota, s)|s \in lI\}$ である。原子状態命題の集合 $P_{\mathrm{ees}}$ は $lP \cup lE$ である。ラベリング関数 $L_{\mathrm{ees}}((e,s))$ は各 $e \in lE$ に対し、$\{e\} \cup lL(s)$ となる。全域的な二項関係の集合 $T_{\mathrm{ees}}$ は $\{((e,s),(e',s'))|e, e \in lE, s, s' \in lS, (s,e',s') \in lT)\}$ である。

$K_{\mathrm{ees}}$ の状態は、$lK$ のイベントと状態とからなる。$K_{\mathrm{ees}}$ のパスは、$lE \times lS$ なる $lK$ の一つの無限列である。$K_{\mathrm{ees}}$ の実行系列とは、$1 \bullet l\pi(0) = \iota$ かつ $2 \bullet l\pi(0) \in lI$ となるパス $l\pi$ のことである。$lK$ のイベントは、$K_{\mathrm{ees}}$ の原子状態命題となる。

それでは、alice と bob が働いている三都市間の移動の例の LKS を、EES-KS として記述したものを $K_{\mathrm{exm3}} \triangleq \langle S_{\mathrm{exm3}}, I_{\mathrm{exm3}}, P_{\mathrm{exm3}}, L_{\mathrm{exm3}}, T_{\mathrm{exm3}}\rangle$ で定め、Maude コードとして記述した形式仕様を提示し、特に 2.3 節で紹介した $K_{\mathrm{exm2}}$ との差分についてそれぞれ説明する。

状態の集合 $S_{\mathrm{exm3}}$ を Maude コードで記述したものは以下の通りである。

```
1  *** S_{exm3} BEGIN
2  fmod PID is
3    sort PersonID .
4    ops alice bob : -> PersonID [ctor] .
5  endfm
6
7  fmod LOCATION is
8    sort Location .
9    ops tokyo osaka fukuoka : -> Location [ctor] .
10 endfm
11
12 *** Labeled Event lE
13 fmod LEVENT is
14   pr PID .
15   sort LEvent .
16   op notran : -> LEvent . *** notran = iota
17   op t2o : PersonID -> LEvent .
18   op o2t : PersonID -> LEvent .
19   op t2f : PersonID -> LEvent .
20   op f2t : PersonID -> LEvent .
21 endfm
22
```

```
fmod SOUP is
  pr PID .
  pr LOCATION .
  pr LEVENT .
  sorts OComp Soup .
  subsort OComp < Soup .
  *** observable components
  op name[_]:_ : PersonID Location -> OComp [ctor] .
  op tran:_ : LEvent -> OComp [ctor] .
  *** soup
  op __ : Soup Soup -> Soup [ctor assoc comm] .
endfm
*** S_{exm3} END
```

Listing 2.9: $S_{\text{exm3}}$

状態の集合 $S_{\text{exm3}}$ には、$S_{\text{exm2}}$ に新しく LEVENT というモジュールを追加した。LEVENT には、ソート LEvent と、その要素のラベル付きイベントが notran、t2o、o2t、t2f、f2t としてそれぞれ定義されている。notran は $\iota$ を示す定数、それ以外は人識別子を受け取ってラベル付きイベントを返す演算として定義される。モジュール SOUP で LEVENT をインポートし、観測可能成分としてのラベル付きイベントが加わったスープとして表現される。

初期状態の集合 $I_{\text{exm3}}$ を Maude コードで記述したものは以下の通りである。

```
*** I_{exm3}
fmod INIT-SOUP is
  pr SOUP .
  op init : -> Soup .
  eq init = (name[alice]: tokyo) (name[bob]: tokyo) (tran:
     notran) .
endfm
```

Listing 2.10: $I_{\text{exm3}}$

初期状態 init は、alice、bob は tokyo に位置し、ラベル付き遷移が notran の、3 つの観測成分からなるスープとして定める。

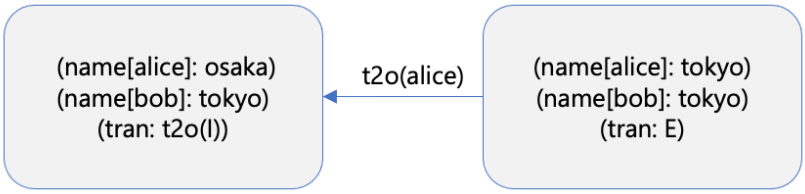


図 2.6: ラベル付き遷移の例

遷移の集合 $T_{\text{exm3}}$ は、以下のような書き換え規則で表現する。

```
42 *** T_{exm3}
43 mod TRANSITION is
44   pr SOUP .
45   var E : LEvent .
46   var I : PersonID .
47   rl [t2o] : (name[I]: tokyo) (tran: E)
48           => (name[I]: osaka) (tran: t2o(I)) .
49   rl [o2t] : (name[I]: osaka) (tran: E)
50           => (name[I]: tokyo) (tran: o2t(I)) .
51   rl [t2f] : (name[I]: tokyo) (tran: E)
52           => (name[I]: fukuoka) (tran: t2f(I)) .
53   rl [f2t] : (name[I]: fukuoka) (tran: E)
54           => (name[I]: tokyo) (tran: f2t(I)) .
55 endm
```

Listing 2.11: $T_{\mathrm{exm3}}$

遷移は、これまでと同様に 4 つの書き換え規則のそれぞれに、ラベル付きイベントの観測成分の書き換えを加えている。例えば、書き換え規則 [t2o] は、人が場所 tokyo から osaka に移り、かつラベル付き遷移が t2o(I) へと変化する、t2o（tokyo to osaka）という名称の遷移であることを表現している。このようにラベル付きイベントを表現する観測可能成分（tran: E）は、直前にどの人が、どのラベル付き遷移を使用したかを状態に記録する役割を持つ。

つづいて、原子状態命題の集合 $P_{\mathrm{exm3}}$ とラベリング関数 $L_{\mathrm{exm3}}$ を Maude コードで記述したものは以下の通りである。

```
56 in model-checker .
57 *** P_{exm3}, L_{exm3}
58 mod PREDICATES is
59   pr TRANSITION .
60   inc SATISFACTION .
61   subsort Soup < State .
62   *** P_{exm3}
63   op inTokyo : PersonID -> Prop .
64   op inOsaka : PersonID -> Prop .
65   op inFukuoka : PersonID -> Prop .
66   op enabled : LEvent -> Prop .
67   op applied : LEvent -> Prop .
68   var I : PersonID .
69   var S : Soup .
70   var PR : Prop .
71   var E : LEvent .
72   *** L_{exm3}
73   eq (name[I]: tokyo) S   |= inTokyo(I) = true .
74   eq (name[I]: osaka) S   |= inOsaka(I) = true .
75   eq (name[I]: fukuoka) S |= inFukuoka(I) = true .
76   eq (name[I]: tokyo) S   |= enabled(t2o(I)) = true .
77   eq (name[I]: tokyo) S   |= enabled(t2f(I)) = true .
```

```
78    eq (name[I]: osaka) S   |= enabled(o2t(I)) = true .
79    eq (name[I]: fukuoka) S |= enabled(f2t(I)) = true .
80    eq (tran: E) S          |= applied(E) = true .
81    eq S |= PR = false [owise] .
82  endm
```

Listing 2.12: $P_{\mathrm{exm3}}$ と $L_{\mathrm{exm3}}$

原子状態命題の集合 $P_{\mathrm{exm3}}$ と $P_{\mathrm{exm2}}$ の差分は、原子状態命題に演算 `enabled` と `applied` を追加したことである。ラベル付きイベント $e$ が実行可能状態であることを enabled($e$) で、$e$ が直前に実行されたことを applied($e$) で表す。これを受け、ラベリング関数 $L_{\mathrm{exm3}}$ は、enabled($e$) と applied($e$) に関する等式を追記して複数行の等式で（ここでは 9 行で）定義する[10]。

## 2.8 公平性の仮定

本節では、具体的な公平性の仮定の定義を提示した後、三都市の移動の例の LKS を EES-KS で記述した Maude コードに対し、公平性を仮定した活性のモデル検査を実施する。

公平性は、さまざまな種類のものがあるが、ここでは弱公平性、強公平性として知られる 2 種類の公平性で記述されたものについて説明する。

**定義 7（ラベル付き遷移 $e$ の弱公平性）** $\Diamond\Box\mathrm{enabled}(e) \Rightarrow \Box\Diamond\mathrm{applied}(e)$

**定義 8（ラベル付き遷移 $e$ の強公平性）** $\Box\Diamond\mathrm{enabled}(e) \Rightarrow \Box\Diamond\mathrm{applied}(e)$

ラベル付き遷移 $e$ の弱公平性は、$lK$ に対し enabled($e$) が不断に（`continuously`）成り立つとき、$e$ がいつかは適用されることを表明する。ある状態遷移 $e$ に対して弱公平性を仮定する、あるいは、ある状態遷移 $e$ を弱公平性で扱う、とは「$e$ が実行可能である状態が、切れ目なく永遠に続くならば $e$ は際限なく実行される、ということを仮定すること」である。$e$ が実行可能状態であることを enabled($e$) で、$e$ が直前に実行されたことを applied($e$) で表すとする。このとき、$e$ がある時点から切れ目なく永遠に実行可能状態であることを $\Diamond\Box\mathrm{enabled}(e)$ で、$e$ は際限なく実行されることを $\Box\Diamond\mathrm{applied}(e)$ として表すことができる。よって、$e$ に対する弱公平性は LTL 式で $\Diamond\Box\mathrm{enabled}(e) \Rightarrow \Box\Diamond\mathrm{applied}(e)$ のように記述できる。

ラベル付き遷移 $e$ の強公平性は、$lK$ に対し enabled($e$) が断続的に（`continually`）成り立つとき、$e$ がいつかは適用されることを表明する。ある状態遷移 $e$ に対して強公平性を仮定する、あるいは、ある状態遷移 $e$ を強公平性で扱う、とは「$e$ が実行

[10] これは例えば、スープ (name[alice] : tokyo) (name[bob] : fukuoka) (tran : t2f(bob)) を受け取り、原子状態命題の集合 {inTokyo(alice), inFukuoka(bob), applied(t2f(bob)), enabled(t2f(alice)), enabled(t2o(alice)), enabled(f2t(bob))} を返すような関数である。

可能である状態が、断続的に続くならば $e$ は際限なく実行される、ということを仮定すること」である。$e$ が実行可能である状態が断続的に続くことを $\Box\Diamond\text{enabled}(e)$ で、$e$ は際限なく実行されることを $\Box\Diamond\text{applied}(e)$ として表すことができる。よって、$e$ に対する強公平性は LTL 式で $\Box\Diamond\text{enabled}(e) \Rightarrow \Box\Diamond\text{applied}(e)$ のように記述できる。

弱公平性を仮定しない場合、$e$ がある時点から切れ目なく永遠に実行可能状態であるにも関わらず、$e$ は際限なく実行されることはない、ということになる。そして、$e$ が無限に実行されないということは、ある時点から $e$ は一切実行されないことを意味する。弱公平性を仮定することで、このように、$e$ が一切実行されないという可能性がある状況、すなわち先の活性のモデル検査で発生した不公平な状況をなくすことができる。強公平性も同様である。

モデル検査用に以下のように定めたモジュール 3CITIES-CHECK について特に 2.4 節との差分について説明する。

```
83  mod 3CITIES-CHECK is
84    pr INIT-SOUP .
85    inc PREDICATES .
86    inc MODEL-CHECKER .
87    inc LTL-SIMPLIFIER .
88    vars P1 P2 : PersonID .
89    var S : Soup .
90    var E : LEvent .
91    op liveness : -> Formula .
92    eq liveness = (<> inFukuoka(alice)) /\ (<> inFukuoka(bob))
          .
93    op wf : LEvent -> Formula .
94    eq wf(E) = (<> [] enabled(E)) -> ([] <> applied(E)) .
95    op wfair : -> Formula .
96    eq wfair = wf(t2o(alice)) /\ wf(t2o(bob)) /\
97               wf(o2t(alice)) /\ wf(o2t(bob)) /\
98               wf(t2f(alice)) /\ wf(t2f(bob)) /\
99               wf(f2t(alice)) /\ wf(f2t(bob)) .
100   op sf : LEvent -> Formula .
101   eq sf(E) = ([] <> enabled(E)) -> ([] <> applied(E)) .
102   op sfair : -> Formula .
103   eq sfair = sf(t2o(alice)) /\ sf(t2o(bob)) /\
104              sf(o2t(alice)) /\ sf(o2t(bob)) /\
105              sf(t2f(alice)) /\ sf(t2f(bob)) /\
106              sf(f2t(alice)) /\ sf(f2t(bob)) .
107 endm
```

Listing 2.13: 三都市間の移動のモデル検査

検査用のモジュール 3CITIES-CHECK では、検査したい性質 liveness に加え、弱公平性を記述した演算 wf(E) と、強公平性を記述した sf(E) を記述した。

弱公平性を意味する wf(E) は、LEVENT E をうけとって<> [] enabled(E)) -> ([] <> applied(E)) という LTL 式を返す。ここで [] は □ で always 演算子、-> は論理包含を表す論理演算子、<>は ◇ で eventually 演算子を意味する。強公平性を意味する sf(E) は、LEVENT E をうけとって [] <> enabled(E)) -> ([] <> applied(E)) という LTL 式を返す。LTL 式 wfair は、ここでは全ての遷移 $e$ に対して弱公平性を仮定している。同様に、sfair は、ここでは全ての遷移 $e$ に対して強公平性を仮定している。

以下のコードを実行することで、実際にモデル検査をする。

```
108 red in 3CITIES-CHECK : modelCheck(init, wfair -> liveness) .
109 red in 3CITIES-CHECK : modelCheck(init, sfair -> liveness) .
```

Listing 2.14: 三都市間の移動のモデル検査の実行

wfair -> liveness で弱公平性を仮定した活性を、sfair -> liveness で強公平性を仮定した活性を表す。なお、->は論理包含を表す論理演算子である。

弱公平性を仮定したモデル検査では、反例が返ってきてモデル検査は失敗する。ここでも、alice は一度も fukuoka へ行かず、tokyo と osaka を無限ループで往復し続けるという反例が検出され失敗してしまう。この例では、弱公平性では不十分なことがわかる。では、強公平性ではどうだろうか。

強公平性を仮定したモデル検査では、以下の結果が返されモデル検査は成功する。

```
reduce in 3CITIES-CHECK : modelCheck(init, sfair -> liveness
   ) .
rewrites: 194322 in 273465ms cpu (389641ms real) (710
   rewrites/second)
result Bool: true
```

Listing 2.15: 三都市間の移動のモデル検査の実行結果

本モデル検査では、実に 194322 回の書き換えと、CPU 時間にして 273465 ミリ秒の実行時間を要する結果となった。

なお、もう一つの例として、以下のような部分的に公平性を仮定したモデル検査をすることもできる。

```
op fairness' : -> Formula .
eq fairness' = sf(t2f(bob)) /\ sf(t2f(alice)) /\
               wf(o2t(bob)) /\ sf(o2t(alice)) .
```

Listing 2.16: 混在して公平性を仮定した三都市間の移動のモデル検査の実行

fairness' は、一部に弱公平性を仮定、一部に強公平性を仮定、その他の遷移

には公平性を仮定しないような、混在した仮定の例である。実行結果は以下の通りである。

```
reduce in VARIATION_TEST : modelCheck(
  init, sf(t2f(bob)) /\ (sf(t2f(alice)) /\
       (wf(o2t(bob)) /\ sf(o2t(alice)))) -> liveness) .
rewrites: 6382 in 105ms cpu (119ms real) (60341 rewrites/
   second)
result Bool: true
```

Listing 2.17: 混在して公平性を仮定した三都市間の移動のモデル検査の実行結果

この場合も、モデル検査は成功する。このように Maude で公平性の仮定した活性の検査をする場合、全ての遷移に対しても可能であるし、あるいは一部に対して弱公平性を、一部に対しては強公平性を、そのほかに関しては公平性を仮定しないように混在させることも可能である。

次章では、三都市間の移動の例に対し、モデル検査の効率を改善する手法である、分割統治アプローチ（DCA）を適応した Maude コードに対してモデル検査を実施する。

## 2.9 分割統治アプローチ（DCA）

前節までで、Maude を用いたモデル検査の一連の流れについて概観した。本節では、2019 年に緒方が提示 [3] した DCA を三都市間の移動の例に適用し、モデル検査の効率化を試みる。

LKS $lK$ の EES-KS $K_{\text{ees}}$ と、公平性 $f_i (i = 1, \ldots, n)$ と活性 $\varphi$ のとき、$(f_1 \wedge \cdots \wedge f_n) \Rightarrow \varphi$ という形の公平性を仮定した活性の LTL 式について考える。

論理式の集合は、連言形の論理式を表したいとき、利便性のためよく利用される [3] 。本節でも利便性のため流用する。具体的には、LTL 式 $\{f_1, \ldots, f_n\} \Rightarrow \varphi$ の中の集合 $\{f_1, \ldots, f_n\}$ は $(f_1 \wedge \cdots \wedge f_n)$ と同じように扱われる。したがって、$\{f_1, \ldots, f_n\} \Rightarrow \varphi$ は、$(f_1 \wedge \cdots \wedge f_n) \Rightarrow \varphi$ と同じ意味として扱われる。

ここで、$j = 1, 2, \ldots, m$ に対し $qf_j$ なる LTL 式と、$F_1 \subseteq \{f_1, f_2, \ldots, f_n\}$ なる部分集合 $F_1$ と、$j' = 1, 2, \ldots, m-1$ に対し $F_{j+1} \subseteq \{f_1, f_2, \ldots, f_n\} \cup \{qf_1, qf_2, \ldots, qf_{j'}\}$ なる部分集合 $F_{j+1}$ で $j = 1, 2, \ldots, m$ に対し $K_{ees} \models F_j \Rightarrow qf_j$ なるものを仮定する。よって、$K_{ees} \models (f_1 \wedge \cdots \wedge f_n) \Rightarrow (qf_1 \wedge \cdots \wedge qf_m)$ となる。このとき、$QF \subseteq \{f_1, f_2, \ldots, f_n\} \cup \{qf_1, qf_2, \ldots, qf_m\}$ をなる部分集合 $QF$ を仮定する。もし $K_{ees} \models QF \Rightarrow \varphi$ ならば $K_{ees} \models (f_1 \wedge \cdots \wedge f_n) \Rightarrow \varphi$ である。したがって、$(f_1 \wedge \cdots \wedge f_n) \Rightarrow \varphi$ は、$F_j \Rightarrow qf_j (i = 1, 2, \ldots, m)$ と $QF \Rightarrow \varphi$ とに分割でき、それぞれに対してモデル検査する、すなわち $K_{ees} \models F_j \Rightarrow qf_j (j = 1, 2, \ldots, m)$ のモデル検査と $K_{ees} \models QF \Rightarrow \varphi$ のモデル検査とをすれば、$K_{ees} \models (f_1 \wedge \cdots \wedge f_n) \Rightarrow$

$\varphi$ モデル検査を満たすことになる。このとき、$qf_j(j = 1, 2, \ldots, m)$ を擬公平性（quasi-fairness）の仮定、と呼ぶ [3]。

モデル検査用に以下のように定めたモジュール 3CITIES-CHECK について、DCA を適用したコードは以下の通りである。

```
87  mod 3CITIES-CHECK is
88    pr INIT-SOUP .
89    inc PREDICATES .
90    inc MODEL-CHECKER .
91    inc LTL-SIMPLIFIER .
92    vars P1 P2 : PersonID .
93    var S : Soup .
94    var E : LEvent .
95    op liveness : -> Formula .
96    eq liveness = (<> inFukuoka(alice)) /\ (<> inFukuoka(bob))
          .
97    op sf : LEvent -> Formula .
98    eq sf(E) = ([] <> enabled(E)) -> ([] <> applied(E)) .
99    op sfair : -> Formula .
100   eq sfair = sf(t2o(alice)) /\ sf(t2o(bob)) /\
101              sf(o2t(alice)) /\ sf(o2t(bob)) /\
102              sf(t2f(alice)) /\ sf(t2f(bob)) /\
103              sf(f2t(alice)) /\ sf(f2t(bob)) .
104   op qfair : -> Formula .
105   eq qfair = sf(t2f(alice)) /\ sf(t2f(bob)) /\
106              sf(o2t(alice)) /\ sf(o2t(bob)) .
107 endm
```

Listing 2.18: 三都市間の移動の DCA によるモデル検査

モジュール 3CITIES-CHECK には、擬公平性として qfair を追記した。公平性を sfair（正確には、全ての遷移に強公平性を仮定したもの）、活性を liveness、論理包含を->とするとき、以下の 3 通りを Maude コードで記述しモデル検査を実施する。

1. 公平性 ⇒ 擬公平性（sfair -> qfair）

2. 擬公平性 ⇒ 活性（qfair -> liveness）

3. 公平性 ⇒ 活性（sfair -> liveness）

上記の 3. を意味する LTL 式のコード sfair -> liveness のモデル検査は、（公平性 ⇒ 擬公平性）⇒ 活性、となることより、1. と 2. をモデル検査することで十分である。この 1. と 2. とに分割し、個別にモデル検査することが、DCA の具体例である。

上記 1. と 2. の実行結果は以下の通りである。

```
reduce in 3CITIES-CHECK : modelCheck(init, sfair -> qfair) .
rewrites: 178097 in 2576ms cpu (2637ms real) (69117 rewrites
   /second)
result Bool: true
reduce in 3CITIES-CHECK : modelCheck(init, qfair -> liveness
   ) .
rewrites: 9073 in 142ms cpu (160ms real) (63671 rewrites/
   second)
result Bool: true
```

Listing 2.19: DCA を適用したモデル検査の実行結果

上記 3. の実行結果は以下の通りである。

```
reduce in 3CITIES-CHECK : modelCheck(init, sfair -> liveness
   ) .
rewrites: 194322 in 273465ms cpu (389641ms real) (710
   rewrites/second)
result Bool: true
```

Listing 2.20: DCA を適用していないモデル検査の実行結果

DCA を適用した 1. と 2. を実行した結果を合算すると、CPU 時間にして約 3 秒であった。DCA を適用していない 3. を実行した結果は、CPU 時間にして約 273 秒であり、かなりの効率化をはかったと言える。

## 2.10 まとめ

本章では、調査対象論文 [3] で提示された「論理式肥大に伴う活性のモデル検査の非効率化を改善するために提案された技術」の一つである DCA を理解するために必要な準備をした。

2.1 節では、Maude を LTL モデル検査器として利用する前段階の最低限の知識を、待ち行列を例に説明した。モデル検査をするためには、対象を何らかの手段で形式化する必要がある。2.2 節では、一番簡単な形式化の例として状態機械を用いて Maude コードで記述したものを用いて、理解を深めた。状態機械は、状態と、状態間の遷移を定義することで、システムの振る舞いを表現できる。以降の節では、三都市間の移動の例を形式化し、実際にモデル検査を実施し実験した。2.3 節では、三都市間の例をクリプケ構造として表現したものを Maude で記述した。クリプケ構造は、原子状態命題と、それに対するラベリング関数を、状態機械に加えて拡張した構造である。クリプケ構造を用いることで、LTL モデル検査が可能となる。次の 2.4 節で、クリプケ構造で定義された LTL 式の構文と、その意味について詳細に説明した。本稿で扱うモデル検査は、活性のモデル検査であるため、LTL 式を用いることで簡潔に表現することができる。ここまでで活性の LTL モデル検

査の準備が整ったため、実際にモデル検査を実施したが、無意味な反例を得て検査は失敗に終わった。この問題は、公平性を仮定した活性のモデル検査で解決することができる。2.5 節と 2.6 節では、公平性を記述するために必要な準備をした。2.5 節では、ラベル付きクリプケ構造の説明した。2.6 節では、ラベル付きクリプケ構造で定義できる SE-LTL 式の構文と意味についてを詳細に定義した。Maude LTL 検査器では、通常のクリプケ構造しか記述することができないため、2.7 節では、LKS を模倣する KS である、EES-KS で三都市間の移動の例を記述した。LKS の SE-LTL 式は、それを模倣する EES-KS の LTL 式で意味が保存され、逆も成り立つ。2.8 節では、三都市間の例の EES-KS に対して、公平性を仮定した LTL モデル検査を行った。この結果、ようやく期待通りの動作をし、反例を得ることなくモデル検査は成功した。三都市間の例は、クリプケ構造も非常に単純であり、プロセス数も 2 個と、非常に小さな例であった。それにも関わらず、公平性を仮定したモデル検査は、CPU 時間にして 273465 ミリ秒という膨大な計算時間をかけなければならなかった。2.9 節では、[3] で緒方が提案した DCA を適用したモデル検査を実施した。この手法を適用した場合、CPU 時間にして 273465 ミリ秒かかっていた検査は、合算して 2718 ミリ秒まで短縮する結果となった。

# 第3章　Qlock相互排除プロトコルに対する無排斥性のモデル検査

本章では、「論理式肥大に伴う活性のモデル検査の非効率化を改善するために提案された技術」であるDCAについて、この技術を用いて解決された事例[3]を再構築して理解を深める。Qlock相互排除プロトコル（Qlock）を例に、DCAを適用してモデル検査を実施し、適用前後で比較する。

最初にQlockを擬似コードを用いて説明し、次にQlockを形式化したクリプケ構造をMaudeで記述する。これに対し、活性の一つである無排斥性（lockout freedom property）のモデル検査を実施すると、極端に偏った反例が検出され失敗してしまう。このような無意味な反例は、公平性（fairness）の仮定を用いることで除外することができる。公平性を仮定した活性のモデル検査は、ラベル付きクリプケ構造を模倣したクリプケ構造（EES-KS）により可能となる。最後に、Qlockに対してDCAを適用したモデル検査を実施し、適用前後で結果を比較する。

## 3.1　Qlock相互排除プロトコル

本節では、はじめにQlockについて概要を示し、擬似コードで形式化したものについて説明する。Qlockは、プロセス識別子のアトミックな待ち行列を利用する相互排除プロトコルである[1]。相互排除とは名前の通り、複数個のプロセスが共有資源を利用する場合に相互に排除しあって、どの時点でも高々1個のプロセスのみその資源を占有することを保証する性質のことである。システムが満たすべき性質は安全性（safety）と活性（liveness）に大別されることは前節でも述べたが、相互排除性は安全性に分類される。本稿では、非決定的選択(non-deterministic choice)な文を用いたものをQlockと呼ぶこととする。これは、ずっと同一の場所に止まり続けるようなプロセスの動作も表現したいからだ。非決定的選択な文を用いたものをQlockとは呼ばない論文がある、ということも付記しておく。

Qlockを、擬似コードで記述したものは以下の通りである。

```
1 Loop
2     "Remainder Section(RS)"
```

[1]待ち行列 (queue) を利用して共用資源に鍵 (lock) をかけるためQlockと呼称される。なお、待ち行列については2.1節に詳しい。

```
3   rs: enq(queue, p_i); | "do something else" and goto rs;
4   ws: repeat until top(queue) = p_i;
5     "Critical Section(CS)"
6   cs: deq(queue);
```

Listing 3.1: Qlock の擬似コード

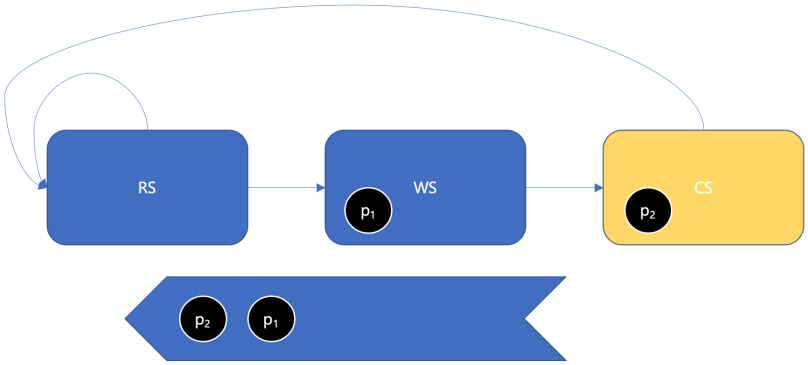


図 3.1: Qlock の例

*queue* は、すべてのプロセス識別子によって共有されるアトミックな待ち行列である。文$_1$| 文$_2$ は、非決定的選択な文で、文$_1$ または 文$_2$ のどちらかが、プロセス $p_i$ により非決定的に実行されることを示す。プロセスは、それぞれ 3 つの場所 `rs` (`remainder section`)、`ws` (`waiting section`)、`cs` (`critical section`) のうちいずれかに存在すると仮定する。コメント文の `CS` ではプロセスは排他的なタスクを処理し、`RS` ではプロセスはその他のタスクを担う。`cs` に入るプロセスは、高々 1 つを許容する。また、`cs` には、いつかはプロセスが 1 つは入ると仮定する。次に、各場所でのプロセスの動作について説明する。

初期状態では、*queue* は空で、各プロセスは `RS`(または `rs`) に位置する。プロセス $p_i$ が `rs` に位置する場合、待ち行列の最後に $p_i$ を追加し更新して `ws` に移動するか、または他の何かを実行して `rs` に止まる。次に、プロセス $p_i$ が `ws` に位置する場合、待ち行列の先頭が $p_i$ になるまでその場で待機する。待ち行列の先頭が $p_i$ のとき、プロセス $p_i$ は `CS`(または `cs`) に入ることができる。最後に、プロセスが `cs` に位置する場合は、待ち行列の先頭は削除され、プロセスは `RS` (または `rs`) に移る。なお、待ち行列は `RS` でも `CS` でも利用されていないと仮定する。

次節では、`Qlock` の擬似コードを形式化したクリプケ構造を `Maude` で記述し、実際にモデル検査を行う。

## 3.2 クリプケ構造によるモデル検査

本節では、Qlock が活性（無排斥性）を満たすことのモデル検査をすることを目的として、Qlock をクリプケ構造で形式化し、それを Maude コードで記述し、最後に Maude の MODEL-CHECKER モジュールと、その関連モジュールを用いて LTL モデル検査を行う。

まず、前節で擬似コードとして形式化した Qlock のクリプケ構造 $K_{\mathrm{Qlock}}$ を $K_{\mathrm{Qlock}} \triangleq \langle S_{\mathrm{Qlock}}, I_{\mathrm{Qlock}}, P_{\mathrm{Qlock}}, L_{\mathrm{Qlock}}, T_{\mathrm{Qlock}} \rangle$ と定め、Maude コードとしてそれぞれ記述した形式仕様を提示し、説明する。

状態の集合 $S_{\mathrm{Qlock}}$ を Maude コードで記述したものは以下の通りである。

```
1  *** S_{Qlock} BEGIN
2  fmod PID is
3    sort Pid .
4    ops p1 p2 : -> Pid [ctor] .
5  endfm
6
7  fmod LOCATION is
8    sort Location .
9    ops rs ws cs : -> Location [ctor] .
10 endfm
11
12 fmod QUEUE is
13   pr PID .
14   sort Queue .
15   op empty : -> Queue [ctor] .
16   op _|_ : Pid Queue -> Queue [ctor] .
17   op enq : Queue Pid -> Queue .
18   op deq : Queue -> Queue .
19   var Q : Queue .
20   vars X Y : Pid .
21   eq enq(empty,X) = X | empty .
22   eq enq(Y | Q,X) = Y | enq(Q,X) .
23   eq deq(empty) = empty .
24   eq deq(X | Q) = Q .
25 endfm
26
27 fmod SOUP is
28   pr PID .
29   pr LOCATION .
30   pr QUEUE .
31   sorts OComp Soup .
32   subsort OComp < Soup .
33   *** observable components
34   op pc[_]:_ : Pid Location -> OComp [ctor] .
35   op queue:_ : Queue -> OComp [ctor] .
36   *** soup
37   op __ : Soup Soup -> Soup [ctor assoc comm] .
```

```
38 endfm
39 *** S_{Qlock} END
```

Listing 3.2: $S_{\mathrm{Qlock}}$

ここでは 4 つのモジュール、PID、LOCATION、QUEUE、SOUP が定義されている。PID はプロセス識別子、LOCATION はプロセスの位置する場所、QUEUE はプロセス識別子の待ち行列の構造、SOUP は状態を観測成分のスープとして記述している。

プロセス識別子 $p_i$ はソート Pid で定義し、場所 rs、ws、cs はソート Location、それぞれの演算に対し ctor 属性を指定して定数として記述する。モジュール SOUP で、先に定めた PID と LOCATION、それと QUEUE をインポートし、ソートとして OComp と Soup を宣言する。subsort キーワードで OComp が Soup のサブソートであることを定め観測可能成分のスープとする。2 章までと同様に、観測可能成分のスープは、juxtaposition 演算子を用いて定め、assoc、comm 属性を指定して結合法則、交換法則を満たすものとする。これにより状態は、プロセスの場所と、待ち行列の状況とを示す観測可能成分の AC スープ（associative-commutative soup）として記述できる。

観測可能成分は、以下の 2 種類を用意する。プロセス識別子 $p_i$ のプロセスが場所 $l$ に位置することを、$\mathrm{pc}[p_i] : l$ という観測可能成分で表現する。これは、Pid と Location を受け取って、OComp を返す中置二項演算子（mixfix operator）である。プロセスが利用する待ち行列の中身が $q$ であることを、$\mathrm{queue} : q$ という観測可能成分で表現する。これは、Queue を受け取って、OComp を返す単項演算子である。

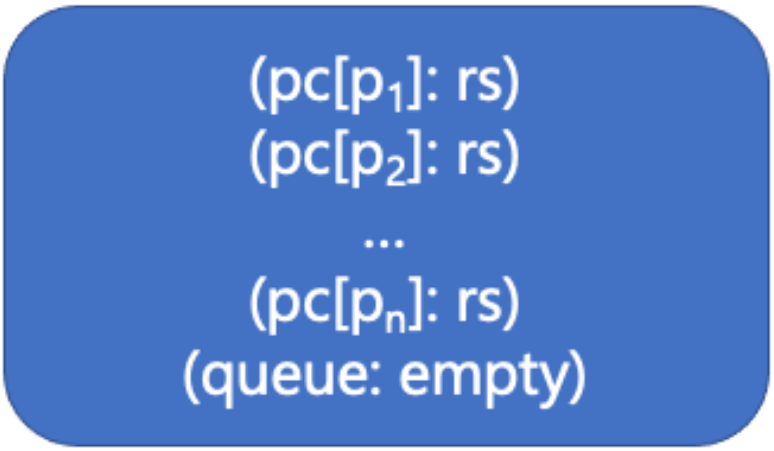


図 3.2: クリプケ構造による Qlock の状態の例

初期状態の集合 $I_{\mathrm{Qlock}}$ を Maude コードで記述したものは以下の通りである。

```
40 *** I_{Qlock}
```

```
41 fmod INIT-SOUP is
42   pr SOUP .
43   op init : -> Soup .
44   eq init = (pc[p1]: rs) (pc[p2]: rs) (queue: empty) .
45 endfm
```

Listing 3.3: $I_{\mathrm{Qlock}}$

$I_{\mathrm{Qlock}}$ は、本節では簡単のためプロセス数を 2 とする。プロセス識別子 $p_1$、$p_2$ のプロセスは全て場所 rs に、待ち行列 $queue$ は空の、観測可能成分のスープ 1 つを初期状態 init として定める。

遷移の集合 $T_{\mathrm{Qlock}}$ は、以下のような書き換え規則で表現する。

```
46 *** T_{Qlock}
47 mod TRANSITION is
48   pr SOUP .
49   var Q : Queue .
50   var I : Pid .
51   rl [dose] : (pc[I]: rs)
52            => (pc[I]: rs) .
53   rl [want] : (pc[I]: rs) (queue: Q)
54            => (pc[I]: ws) (queue: enq(Q,I)) .
55   rl [try]  : (pc[I]: ws) (queue: (I | Q))
56            => (pc[I]: cs) (queue: (I | Q)) .
57   rl [exit] : (pc[I]: cs) (queue: Q)
58            => (pc[I]: rs) (queue: deq(Q)) .
59 endm
```

Listing 3.4: $T_{\mathrm{Qlock}}$

書き換え規則は rl キーワードを用いて記述する。例えば、書き換え規則 [want] は、プロセスが場所 rs から ws に移り、それにともない enq により待ち行列にプロセス識別子が追加される。書き換え規則 [dose] からは、状態の変化がないことが見てとれる。

原子状態命題の集合 $P_{\mathrm{Qlock}}$ と、ラベリング関数 $L_{\mathrm{Qlock}}$ を Maude コードで記述したものは以下の通りである。

```
60 in model-checker .
61 *** P_{ees, Qlock}, L_{ees, Qlock}
62 mod QLOCK-PROP is
63   pr TRANSITION .
64   inc SATISFACTION .
65   subsort Soup < State .
66 *** P_{ees, Qlock}
67   ops wait crit : Pid -> Prop .
68   ops enabled applied : LEvent -> Prop .
```

```
69 *** L_{ees, Qlock}
70   var I : Pid .
71   var Q : Queue .
72   var S : Soup .
73   var PR : Prop .
74   var E : LEvent .
75   eq (pc[I]: ws) S |= wait(I) = true .
76   eq (pc[I]: cs) S |= crit(I) = true .
77   eq (pc[I]: rs) S |= enabled(dose(I)) = true .
78   eq (pc[I]: rs) S |= enabled(want(I)) = true .
79   eq (pc[I]: ws) (queue: (I | Q)) S |= enabled(try(I)) =
       true .
80   eq (pc[I]: cs) S |= enabled(exit(I)) = true .
81   eq (le: E) S   |= applied(E) = true .
82   eq S |= PR = false [owise] .
83 endm
```

Listing 3.5: $P_{\mathrm{Qlock}}$ と $L_{\mathrm{Qlock}}$

はじめに、先に定義したモジュール TRANSITION と、Maude の LTL モデル検査用の model-checker.maude ファイルに定義されているモジュール SATISFACTION をインポートしている。SATISFACTION は、状態に対して原子状態命題の真偽を定めるために用いられる。ソート Soup を、SATISFACTION で定義されているソート State のサブソートとして定めることで、演算 $\_ \mid = \_$ を利用することができる。これは、ソート State と Prop をとってソート Bool を返す演算である。

$P_{\mathrm{Qlock}} \triangleq \{\mathtt{crit}(p_1), \mathtt{crit}(p_2), \mathtt{wait}(p_1), \mathtt{wait}(p_2)\}$ は、上述の $S_{\mathrm{Qlock}}$ で定めたプロセス識別子 Pid を受け取って原子状態命題 Prop を返す 2 つの演算子として定義する。場所 cs にプロセス識別子 $p_i$ が位置することを $\mathtt{crit}(p_i)$ で、場所 ws にプロセス識別子 $p_i$ が位置することを $\mathtt{wait}(p_i)$ で表す。

ラベリング関数 $L_{\mathrm{Qlock}}$ は eq キーワードを用いて複数行の等式で（ここでは 3 行で）定義する。これは、例えばスープ $(\mathrm{pc}[p_1] : \mathrm{ws})\ (\mathrm{pc}[p_2] : \mathrm{cs})\ (\mathrm{queue} : p_2|p_1|\mathit{empty})$ を受け取り、原子状態命題の集合 $\{\mathrm{wait}(p_1),\ \mathrm{crit}(p_2)\}$ を返すような関数である。

以上で、Qlock をクリプケ構造 $K_{\mathrm{Qlock}}$ として Maude で形式化した。次に、これに対し無排斥性のモデル検査をする。無排斥性とは、Qlock が満たすべき性質の一つで、相互排除な処理の際にかけたロックがいつまで経っても開放されないような状況が起こらない性質のことである。先に述べた相互排除性は安全性に分類される一方で、無排斥性は活性に分類される。

無排斥性のモデル検査は、以下の Maude コードで記述できる。

```
84 mod QLOCK-CHECK is
85   pr INIT-SOUP .
86   inc QLOCK-PROP .
87   inc MODEL-CHECKER .
88   inc LTL-SIMPLIFIER .
```

```
89    op lofree : -> Formula .
90    eq lofree = (wait(p1) |-> crit(p1)) /\
91                (wait(p2) |-> crit(p2)) .
92  endm
93  red in QLOCK-CHECK : modelCheck(init, lofree) .
```

Listing 3.6: Qlock の無排斥性のモデル検査

QLOCK-CHECK は、今回のモデル検査のために定義したモジュールで 4 つのモジュールをインポートしている。INIT-SOUP は $I_{\mathrm{Qlock}}$ で、2 つのプロセスが rs にいる状態を示し、QLOCK-PROP は、Qlock のモデル検査の命題を記述しており、MODEL-CHECKER は LTL モデル検査に必須のモジュールで、LTL-SIMPLIFIER はモデル検査の補助モジュールである[2]。今回検査したい性質である無排斥性（lockout freedom property）を形式化した LTL 式は lofree として宣言する。lofree に出現する $|->$ は leadsto 演算子 $\rightsquigarrow$ を、/\ は論理積 $\wedge$ を意味する。よって lofree は、「いつかは wait($p_1$) が crit($p_1$) になり、かつ、いつかは wait($p_2$) が crit($p_2$) になる」という意味である。

最後に、red コマンドで modelCheck() 関数を実行しモデル検査を行う。この関数は、モデル検査用のモジュール MODEL-CHECKER に定義されており、引数として状態と LTL 式をとり、ModelCheckResult を返す。ModelCheckResult は、LTL 式が充足するとき true を、充足しないときは反例が返る。第一引数に初期状態 init、第二引数に無排斥性の LTL 式 lofree を与えて実行した結果は以下の通りである。

```
reduce in QLOCK-CHECK : modelCheck(init, lofree) .
rewrites: 106 in 1ms cpu (1ms real) (100473 rewrites/second)
result ModelCheckResult: counterexample (
  {queue: empty (pc[p1]: rs) pc[p2]: rs,'want},
  {queue: (p1 | empty) (pc[p1]: ws) pc[p2]: rs,'dose})
```

Listing 3.7: Qlock の無排斥性のモデル検査の実行結果

今回の反例は、初期状態 init の遷移名 [want] から始まり、次の状態でプロセス $p_1$ が ws、プロセス $p_2$ が rs に位置し、その次の状態から遷移名 [dose] が選択され続ける、すなわち $p_2$ が rs にずっと止まり続け無限ループをして cs へ入らないというパスを示す。

このようにプロセス $p_2$ ばかりが選ばれて $p_1$ が選ばれないような不公平な反例は、非現実的であり、モデル検査の目的からは有益な情報であるとは言えない。偏った反例を任意に排除する、すなわちシステムの不公平なプロセス割り当てのような望ましくない実行系列を排除してモデル検査をするため、2.8 節で紹介した公平性の仮定を記述する。公平性の仮定を記述するためには、ラベル付きクリプケ

[2]MODEL-CHECKER と LTL-SIMPLIFIER は、先に読み込んだファイル model-checker.maude に定義されている。

構造を模したクリプケ構造を用いる必要がある。次章では、Qlock を拡張し、公平性の仮定を記述できるように修正し、再び無排斥性のモデル検査の実験をする。

## 3.3 EES クリプケ構造によるモデル検査

本節では、前節と同様に Qlock が無排斥性を満たすことのモデル検査をすることを目的とし、公平性を仮定した上で再び検査を行う。公平性の仮定を記述するには、ラベル付きクリプケ構造（LKS）を用いる必要がある。しかしながらラベル付きクリプケ構造そのものは、Maude では記述できない。そこで、状態にイベントを埋め込むことで、通常のクリプケ構造を用いてラベル付きクリプケ構造を模倣したものを、Maude で記述する。この模倣したクリプケ構造は、「ラベル付きクリプケ構造の EES クリプケ構造」と呼ばれ [3]、以降は、LKS の EES-KS（または単に EES-KS）と呼称する。本節の最後に、Qlock の LKS を EES-KS として形式化した Maude コードを用いて、公平性を仮定した無排斥性のモデル検査を再び行う。

それでは、Qlock の EES-KS を $K_{\mathrm{ees,Qlock}}$ で定め、これに対して再び無排斥性のモデル検査を実施する。なお、本節からはプロセス数を増やし、2 個から 8 個までの実験を行う。

まず、$K_{\mathrm{ees,Qlock}} \triangleq \langle S_{\mathrm{ees,Qlock}}, I_{\mathrm{ees,Qlock}}, P_{\mathrm{ees,Qlock}}, L_{\mathrm{ees,Qlock}}, T_{\mathrm{ees,Qlock}} \rangle$ と定め、Maude コードとしてそれぞれ記述した形式仕様を提示し、説明する。

状態の集合 $S_{\mathrm{ees,Qlock}}$ を Maude コードで記述したものは以下の通りである。

```
1  *** S_{ees, Qlock} BEGIN
2  fmod PID is
3    sort Pid .
4    ops p1 p2 p3 p4 p5 p6 p7 p8 : -> Pid [ctor] .
5  endfm
6
7  fmod LOCATION is
8    sort Location .
9    ops rs ws cs : -> Location [ctor] .
10 endfm
11
12 *** Labeled Event lE
13 fmod LEVENT is
14   pr PID .
15   sort LEvent .
16   op notran : -> LEvent . *** notran = iota
17   op dose : Pid -> LEvent .
18   op want : Pid -> LEvent .
19   op try  : Pid -> LEvent .
20   op exit : Pid -> LEvent .
21 endfm
22
23 fmod QUEUE is
```

```
24    pr PID .
25    sort Queue .
26    op empty : -> Queue [ctor] .
27    op _|_ : Pid Queue -> Queue [ctor] .
28    op enq : Queue Pid -> Queue .
29    op deq : Queue -> Queue .
30    var Q : Queue .
31    vars X Y : Pid .
32    eq enq(empty,X) = X | empty .
33    eq enq(Y | Q,X) = Y | enq(Q,X) .
34    eq deq(empty) = empty .
35    eq deq(X | Q) = Q .
36  endfm
37
38  fmod SOUP is
39    pr PID .
40    pr LOCATION .
41    pr QUEUE .
42    pr LEVENT .
43    sorts OComp Soup .
44    subsort OComp < Soup .
45    *** observable components
46    op pc[_]:_ : Pid Location -> OComp [ctor] .
47    op queue:_ : Queue -> OComp [ctor] .
48    op le:_ : LEvent -> OComp [ctor] .
49    *** soup
50    op __ : Soup Soup -> Soup [ctor assoc comm] .
51  endfm
52  *** S_{ees, Qlock} END
```

Listing 3.8: $S_{\mathrm{ees,Qlock}}$

状態の集合 $S_{\mathrm{ees,Qlock}} = \{(e, s)|e \in lE, s \in lS\}$ は、前節で形式化した $S_{\mathrm{Qlock}}$ と同様に、$s$ は、プロセスの場所と、プロセス識別子の待ち行列とを示す観測可能成分とを表す。違いは、モジュール LEVENT を追加したことである。LEVENT には、ソート LEvent と、その要素のラベル付きイベント $e$ が notran、dose、want、try、exit としてそれぞれ定義されている。notran は $\iota$ を示す定数、それ以外はプロセス識別子を受け取ってラベル付きイベントを返す演算 le:_として定義される。これにより、ラベル付きイベント $e$ は観測可能成分として表現され、状態 $(e, s)$ は観測可能成分 4 つからなるスープとして表現される。

初期状態の集合 $I_{\mathrm{ees,Qlock}}$ を Maude コードで記述したものは以下の通りである。

```
53  *** I_{ees, Qlock}
54  fmod INIT-SOUP is
55    pr SOUP .
56    ops init2 init3 init4 init5 init6 init7 init8 : -> Soup .
57    eq init2 = (pc[p1]: rs) (pc[p2]: rs) (queue: empty) (le:
```

```
      notran) .
58    eq init3 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (queue:
      empty) (le: notran) .
59    eq init4 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
       rs) (queue: empty) (le: notran) .
60    eq init5 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
       rs) (pc[p5]: rs) (queue: empty) (le: notran) .
61    eq init6 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
       rs) (pc[p5]: rs) (pc[p6]: rs) (queue: empty) (le:
      notran) .
62    eq init7 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
       rs) (pc[p5]: rs) (pc[p6]: rs) (pc[p7]: rs) (queue:
      empty) (le: notran) .
63    eq init8 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
       rs) (pc[p5]: rs) (pc[p6]: rs) (pc[p7]: rs) (pc[p8]: rs
      ) (queue: empty) (le: notran) .
64  endfm
```

Listing 3.9: $I_{\text{ees,Qlock}}$

$I_{\text{ees,Qlock}} = \{(\iota, s) | s \in lI\}$ は、本節ではプロセスが 2 個から 8 個までの 7 通りを定める。$\iota$ は `notran` で、$s$ は全てのプロセス識別子が場所 `rs`、待ち行列 *empty* で定義する。

遷移の集合 $T_{\text{ees,Qlock}}$ は、以下のような書き換え規則で表現する。

```
65  *** T_{ees, Qlock}
66  mod TRANSITION is
67    pr SOUP .
68    var Q : Queue .
69    var I : Pid .
70    var E : LEvent .
71    rl [dose] : (pc[I]: rs) (le: E)
72              => (pc[I]: rs) (le: dose(I)) .
73    rl [want] : (pc[I]: rs) (queue: Q) (le: E)
74              => (pc[I]: ws) (queue: enq(Q,I)) (le: want(I)) .
75    rl [try]  : (pc[I]: ws) (queue: (I | Q)) (le: E)
76              => (pc[I]: cs) (queue: (I | Q)) (le: try(I)) .
77    rl [exit] : (pc[I]: cs) (queue: Q) (le: E)
78              => (pc[I]: rs) (queue: deq(Q)) (le: exit(I)) .
79  endm
```

Listing 3.10: $T_{\text{ees,Qlock}}$

$T_{\text{ees,Qlock}} = \{((e, s), (e', s')) | e, e \in lE, s, s' \in lS, (s, e', s') \in lT)\}$ は、前節と同様に 4 つの書き換え規則が定められている。ただしそれぞれの遷移に対し、ラベル付きイベントの観測成分の書き換えを加えている。例えば書き換え規則 `[dose]` は、プロセス識別子の位置には変化はないが、ラベル付きイベントの観測可能成分は

(le: dose(I)) となり、直前の遷移が [dose] であることを状態として記録している。

原子状態命題の集合 $P_{\text{ees,Qlock}}$ と、ラベリング関数 $L_{\text{ees,Qlock}}$ を Maude コードで記述したものは以下の通りである。

```
in model-checker .
*** P_{ees, Qlock}, L_{ees, Qlock}
mod QLOCK-PROP is
  pr TRANSITION .
  inc SATISFACTION .
  subsort Soup < State .
*** P_{ees, Qlock}
  op wait : Pid -> Prop .
  op crit : Pid -> Prop .
  op enabled : LEvent -> Prop .
  op applied : LEvent -> Prop .
  var I : Pid .
  var Q : Queue .
  var S : Soup .
  var PR : Prop .
  var E : LEvent .
*** L_{ees, Qlock}
  eq (pc[I]: ws) S |= wait(I) = true .
  eq (pc[I]: cs) S |= crit(I) = true .
  eq (pc[I]: rs) S |= enabled(dose(I)) = true .
  eq (pc[I]: rs) S |= enabled(want(I)) = true .
  eq (pc[I]: ws) (queue: (I | Q)) S |= enabled(try(I)) =
     true .
  eq (pc[I]: cs) S |= enabled(exit(I)) = true .
  eq (le: E) S   |= applied(E) = true .
  eq S |= PR = false [owise] .
endm
```

Listing 3.11: $P_{\text{ees,Qlock}}$ と $L_{\text{ees,Qlock}}$

$P_{\text{ees,Qlock}} = lP \cup lE$ の $lP$ はプロセス識別子 Pid を受け取ってラベル付き原子状態命題 Prop を返す 2 つの演算子 wait、crit として、$lE$ はラベル付き遷移 LEvent を受け取ってラベル付き原子状態命題 Prop を返す 2 つの演算子 enabled、applied として定義する。

$L_{\text{ees,Qlock}} : lS \to \{e\} \cup lL(s)$ は eq キーワードを用いて等式 8 行で定義する。[3] ここで変数 P はプロセス識別子、S は (観測可能成分の) スープ、Q は待ち行列、PR は原子状態命題、E はラベル付きイベントである。

以上で、Qlock をクリプケ構造 $K_{\text{ees,Qlock}}$ として Maude で形式化した。次に、これ

[3] ここで定まるラベリング関数 $L_{\text{ees,Qlock}}$ は、例えばスープ (queue : $(p_1|p_2|empty)$) (pc[$p_1$] : cs) (pc[$p_2$] : ws) (le : try($p_1$)) を受け取り、原子状態命題の集合 {crit($p_1$), wait($p_2$), enabled(exit($p_1$)), enabled(try($p_2$)), applied(try($p_1$))} を返すような関数である。

に対し再び無排斥性のモデル検査をする。無排斥性のモデル検査は、以下の Maude コードで記述できる。

```
106 mod QLOCK-FORMULA is
107   pr INIT-SOUP .
108   inc QLOCK-PROP .
109   inc MODEL-CHECKER .
110   inc LTL-SIMPLIFIER .
111   var E : LEvent .
112   var I : Pid .
113   op wf : LEvent -> Formula .
114   eq wf(E) = (<> [] enabled(E)) -> ([] <> applied(E)) .
115 endm
116
117 mod PROC2MC is
118   pr QLOCK-FORMULA .
119   ops mutex2 lofree2 fair2 qfair2 : -> Formula .
120   eq mutex2 = [] ~(crit(p1) /\ crit(p2)) .
121   eq lofree2 = (wait(p1) |-> crit(p1)) /\ (wait(p2) |-> crit
        (p2)) .
122   eq fair2 = wf(dose(p1)) /\ wf(dose(p2)) /\
123              wf(want(p1)) /\ wf(want(p2)) /\
124              wf(try(p1)) /\ wf(try(p2)) /\
125              wf(exit(p1)) /\ wf(exit(p2)) .
126 endm
127 red in PROC2MC : modelCheck(init2, fair2 -> lofree2) .
```

Listing 3.12: 無排斥性のモデル検査

QLOCK-FORMULA と PROC2MC は、モデル検査のために定義したモジュールである。lofree2 は、検査したい性質である無排斥性を形式化した LTL 式である。wf(E) は、ラベル付きイベントを受け取り、弱公平性の仮定を示す LTL 式を返す。fair2 は、すべてのラベル付きイベントに対して弱公平性を仮定した LTL 式である。wf(E) 内の->は論理包含を表す記号である。したがって、fair2 -> lofree2 は、すべてのラベル付き遷移に対して弱公平性を仮定した無排斥性を記述した LTL 式である。modelCheck() 関数に、第一引数に初期状態 init2、第二引数に fair2 -> lofree2 を与えて実行した結果は以下の通り成功する。

```
reduce in PROC2MC : modelCheck(init2, fair2 -> lofree2) .
rewrites: 16366 in 31496ms cpu (46143ms real) (519 rewrites/
   second)
result Bool: true
```

Listing 3.13: 弱公平性を仮定した無排斥性のモデル検査の実行結果

なお、本稿の実験環境ではプロセス 3 個の場合、一定時間を経過してもモデル

検査は停止しなかった。

次節では、この無排斥性のモデル検査に対して、DCA を適用することで効率化が測れるかどうかについて再現する実験を行い、その効果について分析する。

## 3.4 DCA を適用したモデル検査

$K_{\mathrm{ees,Qlock}}$ に対して無排斥性のモデル検査を、DCA を適用して実行したい。前節で $K_{\mathrm{ees,Qlock}}$ は、全ての遷移に対して弱公平性を仮定したとき、モデル検査は成功した。ここで、try($p_i$) と exit($p_i$) の 2 つの遷移に弱公平性を仮定した場合について考える。これは、プロセス $p_i$ が ws に位置したとき、いつかは必ず cs を通り、rs へ戻ることを意味する。これは、このとき待ち行列に注目すると、プロセス $p_i$ の識別子が先頭に位置し、そしていつかは必ず先頭から削除されることと同義であることが、直感的に想像できる。この待ち行列から必ず削除される性質を dt($p_i$) と表現することとする。dt($p_i$) を定義するために、任意の $p_i \in \mathrm{Pid}$ に対し原子状態命題 $\mathrm{top?}(p_i) \in P_{ees,Qlock}$ を、各 $s_{ees,Qlock} \in S_{ees,Qlock}$ に対して次のように定める。

- $\mathrm{top?}(p_i) \in L_{\mathrm{ees,Qlock}}(s_{\mathrm{ees,Qlock}})$ のとき、かつそのときに限り $(\exists q \in PidQueue)$ $(\mathrm{queue} : p_i \,|\, q) \subseteq s_{\mathrm{ees,Qlock}}$ である。

dt($p_i$) は $\mathrm{top?}(p_i) \rightsquigarrow \neg\mathrm{top?}(p_i)$ と定義できる。qfair は、前節同様にプロセス 2 個の場合の無排斥性のモデル検査のため、$\mathrm{dt}(p_1) \wedge \mathrm{dt}(p_2)$ と定義できる。

- $K_{\mathrm{ees,Qlock}} \models (\mathrm{WF}(\mathrm{try}(p_1)) \wedge (\mathrm{WF}(\mathrm{exit}(p_1)) \Rightarrow \mathrm{dt}(p_1)$
- $K_{\mathrm{ees,Qlock}} \models (\mathrm{WF}(\mathrm{try}(p_2)) \wedge (\mathrm{WF}(\mathrm{exit}(p_2)) \Rightarrow \mathrm{dt}(p_2)$

$K_{\mathrm{ees,Qlock}}$ に擬公平性を追加した Maude コードは以下の通りである。

```
*** P
op top? : Pid -> Prop .
*** L
eq (queue: (I | Q)) S |= top?(I) = true .

*** 擬公平性の仮定
op qfair : -> Formula .
op dt : Pid -> Formula .
eq dt(I) = top?(I) |-> ~ top?(I) .
eq qfair = dt(p1) /\ dt(p2) .
```

Listing 3.14: EES-Qlock に擬公平性の仮定を追加する

擬公平性を仮定して DCA で分割した無排斥性のモデル検査を、実行した結果は以下の通りである。

```
reduce in PROC2MC : modelCheck(init2, wf(try(p1)) /\ wf(exit
   (p1)) -> dt(p1)) .
rewrites: 361 in 2ms cpu (2ms real) (144457 rewrites/second)
result Bool: true
==========================================
reduce in PROC2MC : modelCheck(init2, wf(try(p2)) /\ wf(exit
   (p2)) -> dt(p2)) .
rewrites: 361 in 2ms cpu (3ms real) (128974 rewrites/second)
result Bool: true
==========================================
reduce in PROC2MC : modelCheck(init2, qfair2 -> lofree2) .
rewrites: 521 in 5ms cpu (6ms real) (90940 rewrites/second)
result Bool: true
```

Listing 3.15: プロセス数が 2 個のときの EES_Qlock の DCA による無排斥性のモデル検査の実行結果

公平性 ⇒ 擬公平性のモデル検査は、それぞれ 2 ミリ秒で検査が成功。擬公平性 ⇒ 活性のモデル検査も同様に、5 ミリ秒で検査が成功した。分割しない場合、すなわち公平性 ⇒ 活性のモデル検査は、31496 ミリ秒かかっていたため、大幅に効率化が果たされたと言える。

続けて、プロセス数を増やして効果を確認することとする。プロセス数が 3 個から 7 個までの実行結果は以下の通りである。

```
reduce in PROC3MC : modelCheck(init3, wf(try(p1)) /\ wf(exit
   (p1)) -> dt(p1)) .
rewrites: 928 in 4ms cpu (5ms real) (219437 rewrites/second)
result Bool: true
==========================================
reduce in PROC3MC : modelCheck(init3, wf(try(p2)) /\ wf(exit
   (p2)) -> dt(p2)) .
rewrites: 928 in 3ms cpu (4ms real) (253413 rewrites/second)
result Bool: true
==========================================
reduce in PROC3MC : modelCheck(init3, wf(try(p3)) /\ wf(exit
   (p3)) -> dt(p3)) .
rewrites: 928 in 3ms cpu (3ms real) (250878 rewrites/second)
result Bool: true
==========================================
reduce in PROC3MC : modelCheck(init3, qfair3 -> lofree3) .
rewrites: 2066 in 26ms cpu (29ms real) (77213 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC4MC : modelCheck(init4, wf(try(p1)) /\ wf(exit
   (p1)) -> dt(p1)) .
rewrites: 3965 in 7ms cpu (7ms real) (521779 rewrites/second
   )
```

```
result Bool: true
==========================================
reduce in PROC4MC : modelCheck(init4, wf(try(p2)) /\ wf(exit
   (p2)) -> dt(p2)) .
rewrites: 3965 in 7ms cpu (7ms real) (534654 rewrites/second
   )
result Bool: true
==========================================
reduce in PROC4MC : modelCheck(init4, wf(try(p3)) /\ wf(exit
   (p3)) -> dt(p3)) .
rewrites: 3965 in 7ms cpu (7ms real) (529726 rewrites/second
   )
result Bool: true
==========================================
reduce in PROC4MC : modelCheck(init4, wf(try(p4)) /\ wf(exit
   (p4)) -> dt(p4)) .
rewrites: 3965 in 7ms cpu (7ms real) (552536 rewrites/second
   )
result Bool: true
==========================================
reduce in PROC4MC : modelCheck(init4, qfair4 -> lofree4) .
rewrites: 9125 in 107ms cpu (115ms real) (85262 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC5MC : modelCheck(init5, wf(try(p1)) /\ wf(exit
   (p1)) -> dt(p1)) .
rewrites: 22432 in 37ms cpu (37ms real) (600412 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC5MC : modelCheck(init5, wf(try(p2)) /\ wf(exit
   (p2)) -> dt(p2)) .
rewrites: 22432 in 37ms cpu (37ms real) (602864 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC5MC : modelCheck(init5, wf(try(p3)) /\ wf(exit
   (p3)) -> dt(p3)) .
rewrites: 22432 in 36ms cpu (36ms real) (611492 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC5MC : modelCheck(init5, wf(try(p4)) /\ wf(exit
   (p4)) -> dt(p4)) .
rewrites: 22432 in 36ms cpu (36ms real) (613885 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC5MC : modelCheck(init5, wf(try(p5)) /\ wf(exit
   (p5)) -> dt(p5)) .
```

```
rewrites: 22432 in 37ms cpu (37ms real) (599097 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC5MC : modelCheck(init5, qfair5 -> lofree5) .
rewrites: 48834 in 531ms cpu (583ms real) (91876 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC6MC : modelCheck(init6, wf(try(p1)) /\ wf(exit
   (p1)) -> dt(p1)) .
rewrites: 151513 in 464ms cpu (475ms real) (325922 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC6MC : modelCheck(init6, wf(try(p2)) /\ wf(exit
   (p2)) -> dt(p2)) .
rewrites: 151513 in 408ms cpu (417ms real) (370456 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC6MC : modelCheck(init6, wf(try(p3)) /\ wf(exit
   (p3)) -> dt(p3)) .
rewrites: 151513 in 420ms cpu (424ms real) (359994 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC6MC : modelCheck(init6, wf(try(p4)) /\ wf(exit
   (p4)) -> dt(p4)) .
rewrites: 151513 in 385ms cpu (388ms real) (393210 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC6MC : modelCheck(init6, wf(try(p5)) /\ wf(exit
   (p5)) -> dt(p5)) .
rewrites: 151513 in 388ms cpu (392ms real) (389629 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC6MC : modelCheck(init6, wf(try(p6)) /\ wf(exit
   (p6)) -> dt(p6)) .
rewrites: 151513 in 379ms cpu (382ms real) (398966 rewrites/
   second)
result Bool: true
==========================================
reduce in PROC6MC : modelCheck(init6, qfair6 -> lofree6) .
rewrites: 319848 in 3246ms cpu (3615ms real) (98521 rewrites
   /second)
result Bool: true
==========================================
reduce in PROC7MC : modelCheck(init7, wf(try(p1)) /\ wf(exit
```

```
   (p1)) -> dt(p1)) .
rewrites: 1179896 in 12683ms cpu (12778ms real) (93026
   rewrites/second)
result Bool: true
==========================================
reduce in PROC7MC : modelCheck(init7, wf(try(p2)) /\ wf(exit
   (p2)) -> dt(p2)) .
rewrites: 1179896 in 17317ms cpu (18521ms real) (68131
   rewrites/second)
result Bool: true
==========================================
reduce in PROC7MC : modelCheck(init7, wf(try(p3)) /\ wf(exit
   (p3)) -> dt(p3)) .
rewrites: 1179896 in 13031ms cpu (13081ms real) (90541
   rewrites/second)
result Bool: true
==========================================
reduce in PROC7MC : modelCheck(init7, wf(try(p4)) /\ wf(exit
   (p4)) -> dt(p4)) .
rewrites: 1179896 in 13582ms cpu (13665ms real) (86869
   rewrites/second)
result Bool: true
==========================================
reduce in PROC7MC : modelCheck(init7, wf(try(p5)) /\ wf(exit
   (p5)) -> dt(p5)) .
rewrites: 1179896 in 13010ms cpu (13055ms real) (90688
   rewrites/second)
result Bool: true
==========================================
reduce in PROC7MC : modelCheck(init7, wf(try(p6)) /\ wf(exit
   (p6)) -> dt(p6)) .
rewrites: 1179896 in 12987ms cpu (13033ms real) (90846
   rewrites/second)
result Bool: true
==========================================
reduce in PROC7MC : modelCheck(init7, wf(try(p7)) /\ wf(exit
   (p7)) -> dt(p7)) .
rewrites: 1179896 in 13040ms cpu (13089ms real) (90482
   rewrites/second)
result Bool: true
==========================================
reduce in PROC7MC : modelCheck(init7, qfair7 -> lofree7) .
rewrites: 2488024 in 33040ms cpu (37830ms real) (75302
   rewrites/second)
result Bool: true
```

Listing 3.16: プロセス数を増やした EES_Qlock の DCA による無排斥性のモデル検査の実行結果

プロセス数を 7 個まで増やした場合は、正常にモデル検査を終えた。プロセス

数が８個の時、スタックオーバーフローが発生し、モデル検査を実施できなかった。以上までの結果をまとめたものは、図 3.3 の表である。

| プロセス(個) | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| DCA (ミリ秒) | 9 | 36 | 135 | 714 | 5,690 | 128,686 | NA(stack overflaw) |
| non-DCA (ミリ秒) | 31,496 | NA | NA | NA | NA | NA | NA |

図 3.3: Qlock の DCA 適用前後の比較

DCA 適用前では、実験環境ではプロセス２個までしか結果を得ることができなかった。対して、DCA 適用後は、プロセス７個に拡張しても時間内に結果を得ることができた。プロセス８個の場合はスタックオーバーフローが発生しモデル検査が正常に終了しなかった。これは、モデル検査そのものの問題である、状態空間爆発に遭遇したためと考えられる。

以上より、活性のモデル検査における DCA は、かなりの効率化に寄与する技術であることが確かめられた。

## 3.5 まとめ

本章では、活性のモデル検査における効率化を図る技術である DCA について、Qlock を題材に実験をし、その手法と効果について理解を深めてきた。3.1 節では、Qlock に対し擬似コードを用いて仕様を与えた。本稿で扱った Qlock は、相互排除性というシステムの安全性は保証するが、無排斥性というシステムの活性は（すなおには）保証しないものであった。活性は、LTL 式で記述することができ、LTL 式はクリプケ構造で扱うことができる。3.2 節では、Qlock の擬似コードを形式化したクリプケ構造を、Maude で記述した。クリプケ構造は状態機械を拡張した構造であり、状態を扱うことができる。Qlock では、プロセスの場所と、待ち行列の状況とを示す観測可能成分の AC スープとして記述した。以上までで、モデル検査を実行する準備は整ったが、このままなんの工夫もなく実行すると、プロセスは期待通りの振る舞いをせず無限ループをしてしまい検査は失敗する。このような無意味な反例は、公平性を仮定することで、回避することができた。公平性を記述するためには、ラベル付きクリプケ構造を用いる必要があったため、3.3 節では、Qlock を EES-KS で形式化して再びモデル検査を実施した。全ての遷移に対して弱公平性を仮定した Qlock は、反例を返すことなく検査を終え、無排斥性を保証することとなり、モデル検査としては一応の成功を見せた。しかしながら、公平性を仮定したモデル検査は、その LTL 式が爆発的に大きくなることから、実行時間に対する非効率を改善する余地があった。3.4 節では、緒方が提示した手法 [3] である DCA を Qlock に適用し、実行時間の効率化がどの程度図れるかを検証

した。結果は、プロセス２個の場合の実験では、DCA を適用する前が 31496 ミリ秒かかっていたことに対し、DCA 適用後は、合算して 9 ミリ秒まで短縮する結果となった。また、プロセス数を増やした場合、当該アプローチを適用しなかった場合は結果を得られなかったことに対し、適用後は一定時間内に結果を得ることが確認できた。

# 第4章　TAS相互排除プロトコルに対する無排斥性のモデル検査

本章では、もう一つの例としてTAS相互排除プロトコル（TAS）を題材として、「論理式肥大に伴う活性のモデル検査の非効率化を改善するために提案された技術」であるDCAの理解を深める。はじめにTASを擬似コードを用いて説明し、次にTASを形式化したEES-KSをMaudeで記述する。これに対し、活性の一つである無排斥性のモデル検査を実施すると、無意味な反例が検出され失敗してしまうことを確認し、公平性を仮定する必要があることを示す。最後に、TASに対してDCAを適用したモデル検査を実施し、プロセス数と、適用前後で結果の比較と考察を示す。

## 4.1　TAS相互排除プロトコル

本節では、はじめにTASについて概要を示し、擬似コードで形式化したものについて説明する。TASは、コンピュータの命令の一つであるtest&setを用いた相互排除プロトコルである。待ち行列を利用して共用資源に鍵をかけるQlockとは違い、全てのプロセスは順番に関係なく、共有資源である鍵の状態を確認することができる。鍵がかかっていれば何もせず、鍵がかかっていなければ施錠してcsへ移動し、目的の処理を排他的に実行できる。処理が終われば、鍵を解錠しrsへ移動する。このように、一つの鍵を利用することで、TASは排他制御を実現する。

TASを、擬似コードで記述したものは以下の通りである。

```
1  Loop
2    rs: "Remainder Section(RS)"
3    ws: repeat while test&set(locked);
4      "Critical Section(CS)"
5    cs: locked := false;
```

Listing 4.1: TASの擬似コード

*locked*は、すべてのプロセス識別子によって共有される鍵で真理値を返す。プロセスは、それぞれ3つの場所rs（remainder section）、ws（waiting section）、cs（critical section）のうちいずれかに存在すると仮定する。コメント文のCSではプロセスは排他的なタスクを処理し、RSではプロセスはその他のタスクを担

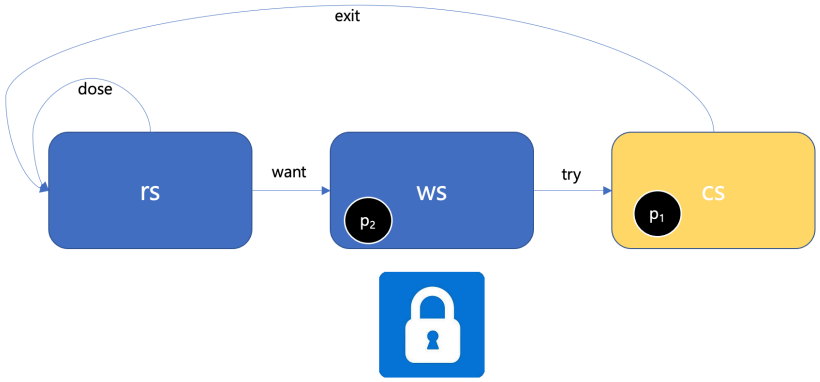


図 4.1: TAS の例

う。cs に入るプロセスは、高々 1 つを許容する。test&set($x$) は、$x$ が false なら $x$ の値を true にすると共に fasle を返し、そうでなければ true を返すことを不可分 (atomic) に行う。次に、各場所でのプロセスの動作について説明する。

初期状態では、$locked$ は false で、各プロセスは rs に位置する。プロセス $p_i$ が rs に位置する場合、ws に移動するか、または他の何かを実行して rs に止まる。次に、プロセス $p_i$ が ws に位置する場合、$locked$ が false になるまでその場で待機する。$locked$ が true のとき、プロセスは CS(または cs) に入ることができる。最後に、プロセスが cs に位置する場合は、$locked$ を false にして、プロセスは rs に移る。なお、$locked$ は RS でも CS でも利用されていないと仮定する。

次節では、TAS の擬似コードを形式化した EES-KS を Maude で記述し、実際にモデル検査を行う。

## 4.2 EES クリプケ構造によるモデル検査

本節では、TAS を EES-KS で形式化し、それを Maude コードで記述し、最後に Maude の MODEL-CHECKER モジュールと、その関連モジュールを用いて活性（無排斥性）の LTL モデル検査を行う。

まず、前節で擬似コードとして形式化した TAS の EES-KS $K_{\mathrm{ees,TAS}}$ を $K_{\mathrm{ees,TAS}} \triangleq \langle S_{\mathrm{ees,TAS}}, I_{\mathrm{ees,TAS}}, P_{\mathrm{ees,TAS}}, L_{\mathrm{ees,TAS}}, T_{\mathrm{ees,TAS}} \rangle$ と定め、Maude コードとしてそれぞれ記述した形式仕様を提示し、説明する。

状態の集合 $S_{\mathrm{ees,TAS}}$ を Maude コードで記述したものは以下の通りである。

```
1 *** S_{ees, TAS} BEGIN
2 fmod PID is
3   sort Pid .
4   ops p1 p2 p3 p4 p5 p6 p7 p8 : -> Pid [ctor] .
5 endfm
6
```

```
 7 fmod LOCATION is
 8   sort Location .
 9   ops rs ws cs : -> Location [ctor] .
10 endfm
11
12 *** Labeled Event lE
13 fmod LEVENT is
14   pr PID .
15   sort LEvent .
16   op notran : -> LEvent . *** notran = iota
17   ops dose want try exit : Pid -> LEvent .
18 endfm
19
20 fmod LOCK is
21   sort Lock .
22   subsort Lock < Bool .
23 endfm
24
25 fmod SOUP is
26   pr PID .
27   pr LOCATION .
28   pr LOCK .
29   pr LEVENT .
30   sorts OComp Soup .
31   subsort OComp < Soup .
32   *** observable components
33   op pc[_]:_ : Pid Location -> OComp [ctor] .
34   op locked:_ : Bool -> OComp [ctor] .
35   op le:_ : LEvent -> OComp [ctor] .
36   *** soup
37   op __ : Soup Soup -> Soup [ctor assoc comm] .
38 endfm
39 *** S_{ees, TAS} END
```

Listing 4.2: $S_{\mathrm{ees,TAS}}$

状態の集合 $S_{\mathrm{ees,TAS}} = \{(e, s) | e \in lE, s \in lS\}$ は、$s$ は、プロセスの場所と、プロセス識別子の待ち行列とを示す観測可能成分とを表す。LEVENT には、ソート LEvent と、その要素のラベル付きイベント $e$ が notran、dose、want、try、exit としてそれぞれ定義されている。notran は $\iota$ を示す定数、それ以外はプロセス識別子を受け取ってラベル付きイベントを返す演算 le:_として定義される。これにより、ラベル付きイベント $e$ は観測可能成分として表現され、状態 $(e, s)$ はプロセスの位置、鍵の状態、直前の遷移、を観測可能な成分からなるスープとして表現される。

初期状態の集合 $I_{\mathrm{ees,TAS}}$ を Maude コードで記述したものは以下の通りである。

```
40 *** I_{ees, TAS}
41 fmod INIT-SOUP is
42   pr SOUP .
```

```
43   ops init2 init3 init4 init5 init6 init7 init8 : -> Soup .
44   eq init2 = (pc[p1]: rs) (pc[p2]: rs) (locked: false) (le:
       notran) .
45   eq init3 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (locked:
        false) (le: notran) .
46   eq init4 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
        rs) (locked: false) (le: notran) .
47   eq init5 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
        rs) (pc[p5]: rs) (locked: false) (le: notran) .
48   eq init6 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
        rs) (pc[p5]: rs) (pc[p6]: rs) (locked: false) (le:
       notran) .
49   eq init7 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
        rs) (pc[p5]: rs) (pc[p6]: rs) (pc[p7]: rs) (locked:
       false) (le: notran) .
50   eq init8 = (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]:
        rs) (pc[p5]: rs) (pc[p6]: rs) (pc[p7]: rs) (pc[p8]: rs
       ) (locked: false) (le: notran) .
51 endfm
```

Listing 4.3: $I_{\text{ees,TAS}}$

$I_{\text{ees,TAS}} = \{(\iota, s) | s \in lI\}$ は、本節ではプロセスが 2 個から 8 個までの 7 通りを定める。$\iota$ は `notran` で、$s$ は全てのプロセス識別子が場所 `rs`、*locked* は `false` で定義する。

遷移の集合 $T_{\text{ees,TAS}}$ は、以下のような書き換え規則で表現する。

```
52 *** T_{ees, TAS}
53 mod TRANSITION is
54   pr SOUP .
55   var B : Lock .
56   var I : Pid .
57   var E : LEvent .
58   rl [dose] : (pc[I]: rs) (le: E)
59            => (pc[I]: rs) (le: dose(I)) .
60   rl [want] : (pc[I]: rs) (le: E)
61            => (pc[I]: ws) (le: want(I)) .
62   rl [try]  : (pc[I]: ws) (locked: false) (le: E)
63            => (pc[I]: cs) (locked: true) (le: try(I)) .
64   rl [exit] : (pc[I]: cs) (locked: true) (le: E)
65            => (pc[I]: rs) (locked: false) (le: exit(I)) .
66 endm
```

Listing 4.4: $T_{\text{ees,TAS}}$

$T_{\text{ees,TAS}} = \{((e, s), (e', s')) | e, e \in lE, s, s' \in lS, (s, e', s') \in lT)\}$ は、4 つの書き換え規則が定められている。例えば書き換え規則 `[dose]` は、プロセス識別子の位置には変化はないが、ラベル付きイベントの観測可能成分は `(le: dose(I))` とな

り、直前の遷移が [dose] であることを状態として記録している。

原子状態命題の集合 $P_{\text{ees,TAS}}$ と、ラベリング関数 $L_{\text{ees,TAS}}$ を Maude コードで記述したものは以下の通りである。

```
67 in model-checker .
68 mod TAS-PROP is
69   pr TRANSITION .
70   inc SATISFACTION .
71   subsort Soup < State .
72 *** P_{ees, TAS}
73   ops wait crit : Pid -> Prop .
74   ops enabled applied : LEvent -> Prop .
75 *** L_{ees, TAS}
76   var I : Pid .
77   var S : Soup .
78   var PR : Prop .
79   var E : LEvent .
80   eq (pc[I]: ws) S |= wait(I) = true .
81   eq (pc[I]: cs) S |= crit(I) = true .
82   eq (pc[I]: rs) S |= enabled(dose(I)) = true .
83   eq (pc[I]: rs) S |= enabled(want(I)) = true .
84   eq (pc[I]: ws) (locked: false) S |= enabled(try(I)) = true
       .
85   eq (pc[I]: cs) S |= enabled(exit(I)) = true .
86   eq (le: E) S   |= applied(E) = true .
87   eq S |= PR = false [owise] .
88 endm
```

Listing 4.5: $P_{\text{ees,TAS}}$ と $L_{\text{ees,TAS}}$

$P_{\text{ees,TAS}} = lP \cup lE$ の $lP$ はプロセス識別子 Pid を受け取ってラベル付き原子状態命題 Prop を返す 2 つの演算子 wait、crit として、$lE$ はラベル付き遷移 LEvent を受け取ってラベル付き原子状態命題 Prop を返す 2 つの演算子 enabled、applied として定義する。

$L_{\text{ees,TAS}} : lS \to \{e\} \cup lL(s)$ は eq キーワードを用いて等式 8 行で定義する。[1] ここで変数 I はプロセス識別子、S は (観測可能成分の) スープ、PR は原子状態命題、E はラベル付きイベントである。

以上で、TAS をクリプケ構造 $K_{\text{ees,TAS}}$ として Maude で形式化した。次に、これに対し無排斥性のモデル検査をする。無排斥性のモデル検査は、以下の Maude コードで記述できる。

```
89 mod TAS-FORMULA is
```

[1] ここで定まるラベリング関数 $L_{\text{ees,TAS}}$ は、例えばスープ (locked : $true$) (pc[$p_1$] : cs) (pc[$p_2$] : ws) (le : try($p_1$) を受け取り、原子状態命題の集合 {crit($p_1$), wait($p_2$), enabled(exit($p_1$)), applied(try($p_1$))} を返すような関数である。

```
  pr INIT-SOUP .
  inc TAS-PROP .
  inc MODEL-CHECKER .
  inc LTL-SIMPLIFIER .
  var I : Pid .
  var E : LEvent .
  ops wf sf : LEvent -> Formula .
  ops lofree : Pid -> Formula .
  eq wf(E) = (<> [] enabled(E)) -> ([] <> applied(E)) .
  eq sf(E) = ([] <> enabled(E)) -> ([] <> applied(E)) .
  eq lofree(I) = wait(I) |-> crit(I) .
endm

mod PROC2MC is
  pr TAS-FORMULA .
  var S : Soup .
  ops lofree2 wfair2 sfair2 qfair2 : -> Formula .
  eq lofree2 = lofree(p1) /\ lofree(p2) .
  eq wfair2 = wf(dose(p1)) /\ wf(dose(p2)) /\
              wf(want(p1)) /\ wf(want(p2)) /\
              wf(try(p1)) /\ wf(try(p2)) /\
              wf(exit(p1)) /\ wf(exit(p2)) .
  eq sfair2 = sf(dose(p1)) /\ sf(dose(p2)) /\
              sf(want(p1)) /\ sf(want(p2)) /\
              sf(try(p1)) /\ sf(try(p2)) /\
              sf(exit(p1)) /\ sf(exit(p2)) .
endm
```

Listing 4.6: 無排斥性のモデル検査

TAS-FORMULA と PROC2MC は、モデル検査のために定義したモジュールである。wf(E)、sf(E) は、ラベル付きイベントを受け取り、それぞれ弱公平性の仮定と、強公平性の仮定を示す LTL 式を返す。lofree(I) は、プロセス識別子を受け取り、無排斥性を示す LTL 式を返す。これにより、プロセス 2 個の場合の無排斥性は、lofree2 で定められる。wfair2 はすべてのラベル付きイベントに対して弱公平性を、sfair2 はすべてのラベル付きイベントに対して強公平性を仮定した LTL 式である。

以上で、無排斥性のモデル検査の準備が整った。これに対し、以下のコマンドでモデル検査を実行する。

```
reduce in PROC2MC : modelCheck(init2, lofree2) .
reduce in PROC2MC : modelCheck(init2, wfair2 -> lofree2) .
reduce in PROC2MC : modelCheck(init2, sfair2 -> lofree2) .
```

Listing 4.7: 無排斥性のモデル検査の実行

公平性を一切仮定しない modelCheck(init2, lofree2) は、反例を得て失敗す

る。一方のプロセスが ws に止まり続け、もう片方のプロセスのみ何度も cs を通過する結果となる。全ての遷移に対して弱公平性を仮定した modelCheck(init2, wfair2 -> lofree2) も、反例を得て失敗する。同様に一方のプロセスが ws に止まり続け、もう片方のプロセスのみ何度も cs を通過する結果となる。これは、弱公平性では仮定が弱いためである。全ての遷移に対して強公平性を仮定した modelCheck(init2, sfair2 -> lofree2) は、成功する。結果は、CPU 時間にして 691799 ミリ秒で、195666 回の書き換えとなった。プロセス 3 個に増やした場合は、半日経っても停止しない結果となった。

次節では、この無排斥性のモデル検査に対して、DCA を適用することで効率化が測れるかどうかについて実験を行い、その効果について分析する。

## 4.3 DCA を適用したモデル検査

前節で $K_{\mathrm{ees,TAS}}$ は、プロセス 2 個の場合の実験では、全ての遷移に対して強公平性を仮定することでモデル検査は成功した。しかしながら、プロセス 3 個の実験では、一定時間内にモデル検査は停止しなかった。本節では、$K_{\mathrm{ees,TAS}}$ に対し、プロセス数 2 個から 8 個までの場合を DCA を適用した無排斥性のモデル検査をし、その効果について検証したい。

遷移を利用した DCA では、どの遷移に対して、どの公平性を仮定すれば良いかの一般的な解は、本稿執筆時点ではまだ得られていない。本節では、2019 年に緒方が証明した分割方法 [3] に当てはまるよう、適当に分割したものに対してモデル検査を行い確認する。

プロセス $p_1$ とその他のプロセス $p_2$ と $p_3$ について、無排斥性が成り立たない場合を考える。プロセス $p_1$ は、無排斥性の前提から ws に位置するとする。かつ、$p_2$ が cs 位置するとき、無排斥性が成り立たないためには $p_3$ のみに実行割当されれば良いことが直感的にわかる。これを排するため、$p_1$ 以外のプロセスは、cs に位置したときに必ず rs へ移動する制約をつければよい。これは遷移 exit に弱公平性を仮定することで表現できる。これで、プロセス $p_1$ 以外の無排斥性は示された。あとは、プロセス $p_1$ が、ws から cs へ移動すれば無排斥性が成り立つと予想されるため、これを遷移 try に $p_1$ の強公平性を仮定してやれば良いことが推測できる。これを、充足関係として表現したものは以下の通りである。

- $K_{\mathrm{ees,TAS}} \models (\mathrm{SF}(\mathrm{try}(p_1)) \wedge (\mathrm{WF}(\mathrm{exit}(p_2)) \wedge (\mathrm{WF}(\mathrm{exit}(p_3)) \Rightarrow \mathrm{lofree}(p_1)$
- $K_{\mathrm{ees,TAS}} \models (\mathrm{SF}(\mathrm{try}(p_2)) \wedge (\mathrm{WF}(\mathrm{exit}(p_3)) \wedge (\mathrm{WF}(\mathrm{exit}(p_1)) \Rightarrow \mathrm{lofree}(p_2)$
- $K_{\mathrm{ees,TAS}} \models (\mathrm{SF}(\mathrm{try}(p_3)) \wedge (\mathrm{WF}(\mathrm{exit}(p_1)) \wedge (\mathrm{WF}(\mathrm{exit}(p_2)) \Rightarrow \mathrm{lofree}(p_3)$
- $K_{\mathrm{ees,TAS}} \models \mathrm{lofree}(p_1) \wedge \mathrm{lofree}(p_2) \wedge \mathrm{lofree}(p_3) \Rightarrow \mathrm{lofree}(p_1) \wedge \mathrm{lofree}(p_2) \wedge \mathrm{lofree}(p_3)$

上記４式中、上から３つまでの式の lofree($p_i$) は、擬公平性を表しており、lofree($p_i$) = wait($p_i$) $\rightsquigarrow$ crit($p_i$) である。今回の分割では、上記の４番目の LTL 式から確認できるように恒真式であり、全体として真であることが見て取れるため、モデル検査は省略することとする。$K_{\mathrm{ees,TAS}}$ に対して、DCA を適用した Maude コードと、実行結果は以下の通りである。

```
reduce in TAS-FORMULA : modelCheck(init3, wf(exit(p3)) /\ (
   wf(exit(p2)) /\ sf(try(p1))) -> lofree(p1)) .
rewrites: 1926 in 18ms cpu (21ms real) (102218 rewrites/
   second)
result Bool: true
==========================================
reduce in TAS-FORMULA : modelCheck(init3, wf(exit(p1)) /\ (
   wf(exit(p3)) /\ sf(try(p2))) -> lofree(p2)) .
rewrites: 1926 in 19ms cpu (22ms real) (96905 rewrites/
   second)
result Bool: true
==========================================
reduce in TAS-FORMULA : modelCheck(init3, wf(exit(p2)) /\ (
   wf(exit(p1)) /\ sf(try(p3))) -> lofree(p3)) .
rewrites: 1926 in 20ms cpu (22ms real) (93572 rewrites/
   second)
result Bool: true
```

Listing 4.8: DCA を適用したプロセス３個の場合のモデル検査実行結果

公平性 $\Rightarrow$ 擬公平性のモデル検査は、それぞれ 20 ミリ秒程度で検査が成功。今回は、擬公平性 $\Rightarrow$ 活性のモデル検査は、恒真のため不要であった。DCA を適用しない場合、すなわち公平性 $\Rightarrow$ 活性のモデル検査は、半日経っても停止しなかったため、大幅に効率化が果たされたと言える。

改めて、プロセス２個の場合からプロセス８個までの、公平性 $\Rightarrow$ 擬公平性のモデル検査を、各プロセスにつき１つずつ検査した結果は以下の通りである。[2]

```
reduce in PROC2MC : modelCheck(init2, wf(exit(p2)) /\ sf(try
   (p1)) -> lofree(p1)) .
rewrites: 738 in 5ms cpu (6ms real) (126782 rewrites/second)
result Bool: true
==========================================
reduce in TAS-FORMULA : modelCheck(init3, wf(exit(p3)) /\ (
   wf(exit(p2)) /\ sf(try(p1))) -> lofree(p1)) .
rewrites: 1926 in 15ms cpu (18ms real) (124322 rewrites/
   second)
result Bool: true
==========================================
reduce in TAS-FORMULA : modelCheck(init4, wf(exit(p4)) /\ (
```

[2]公平性 $\Rightarrow$ 活性のモデル検査は、恒真のため不要である。

```
   wf(exit(p3)) /\ (wf(exit(p2)) /\ sf(try(p1)))) -> lofree(
   p1)) .
rewrites: 4935 in 50ms cpu (62ms real) (97595 rewrites/
   second)
result Bool: true
==========================================
reduce in TAS-FORMULA : modelCheck(init5, wf(exit(p5)) /\ (
   wf(exit(p4)) /\ (wf(exit(p3)) /\ (wf(exit(p2)) /\ sf(try(
   p1))))) -> lofree(p1)) .
rewrites: 12634 in 210ms cpu (273ms real) (60117 rewrites/
   second)
result Bool: true
==========================================
reduce in TAS-FORMULA : modelCheck(init6, wf(exit(p6)) /\ (
   wf(exit(p5)) /\ (wf(exit(p4)) /\ (wf(exit(p3)) /\ (wf(
   exit(p2)) /\ sf(try(p1)))))) -> lofree(p1)) .
rewrites: 32405 in 811ms cpu (1098ms real) (39948 rewrites/
   second)
result Bool: true
==========================================
reduce in TAS-FORMULA : modelCheck(init7, wf(exit(p7)) /\ (
   wf(exit(p6)) /\ (wf(exit(p5)) /\ (wf(exit(p4)) /\ (wf(
   exit(p3)) /\ (wf(exit(p2)) /\ sf(try(p1)))))) -> lofree(
   p1)) .
rewrites: 83046 in 3873ms cpu (5541ms real) (21440 rewrites/
   second)
result Bool: true
==========================================
reduce in TAS-FORMULA : modelCheck(init8, wf(exit(p8)) /\ (
   wf(exit(p7)) /\ (wf(exit(p6)) /\ (wf(exit(p5)) /\ (wf(
   exit(p4)) /\ (wf(exit(p3)) /\ (wf(exit(p2)) /\ sf(try(
p1)))))))) -> lofree(p1)) .
rewrites: 211855 in 20803ms cpu (30115ms real) (10183
   rewrites/second)
result Bool: true
```

Listing 4.9: DCA を適用したプロセス 2 から 8 個までのモデル検査実行結果

今回の TAS の DCA の実験では、プロセス数の場合毎に 1 つずつ検査してある。したがって、CPU 実行時間を比較する場合は、実行時間をプロセス数で乗じたものが分割統治全体の結果となる。以上までの結果をまとめたものは、図**??**である。

DCA 適用前では、実験環境ではプロセス 2 個までしか結果を得ることができなかった。対して、DCA 適用後は、プロセス 8 個に拡張しても一定時間内に結果を得ることができた。

| プロセス(個) | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|
| DCA (ミリ秒) | 10 | 45 | 200 | 1,050 | 4,866 | 27,111 | 166,424 |
| non-DCA (ミリ秒) | 691,799 | NA | NA | NA | NA | NA | NA |

図 4.2: TAS の DCA 適用前後の比較

## 4.4 まとめ

本章では、活性のモデル検査における効率化を図る技術である DCA について、3 章の Qlock に引き続き、TAS を題材に実験をし、その手法と効果について理解を深めてきた。4.1 節では、TAS に対し擬似コードを用いて仕様を与えた。本稿で扱った TAS は、公平性の仮定なしには無排斥性のモデル検査をすることができないため、4.2 節で EES-KS を用いて Maude コードで形式化した。EES-KS は活性を記述することができる LTL 式を扱うことができ、さらに公平性を記述できる LK を模倣するクリプケ構造である。TAS の活性を検査するためには弱公平性では不十分であり、全ての遷移に強公平性を仮定することで十分であった。公平性を仮定したモデル検査は、その LTL 式が爆発的に大きくなることから、本稿の実行環境では、プロセス数 2 個までの検査はできたものの、3 個の場合は一定時間内に検査を終えることができなかった。そこで 4.3 節では、緒方が提示した手法 [3] である DCA を TAS に適用し、実行時間の効率化がどの程度図れるかを検証した。結果は、プロセス 2 個の場合の実験では、DCA を適用する前が 691799 ミリ秒かかっていたことに対し、DCA 適用後は、合算して 10 ミリ秒まで短縮する結果となった。また、プロセス数を 8 個まで増やした場合も、一定時間内に検査を終えることができることが確認できた。

# 第5章　関連研究

本章では、活性のモデル検査で必要となる、公平性の仮定をサポートした主だったモデル検査器の研究について調査した。その特徴について、本稿で用いたMaude LTLモデル検査器やその他の検査器と比較をしつつ、２つの関連研究について報告する。なお、章末には、2021年1月13日時点のモデル検査器について、公平性の利用の可否を主眼に、簡単に比較したテーブルを掲載する。

## 5.1　PAT(Process Analysis Toolkit)

PAT（Process Analysis Toolkit）は、シンガポール国立大学で研究開発されている、CSP（Communicating Sequential Process）などのプロセス代数を基本計算モデルとした、並行システムなどのモデル検査を可能とするソフトウェアツールである [6]。SALのような公平性を対象としない検査器や、弱公平性のみをサポートしているSPINモデル検査器とは異なり、強公平性を仮定した活性のモデル検査が可能である。PATは、検査の表現力を一定程度制限し、専用のアルゴリズムを用いることで公平性の仮定を実現している。PATは、二通りの方法で公平性を仮定することができる [12]。一つはコードのアノテーションによる方法で、もう一方はプルダウン式の選択肢から選ぶ方法である。アノテーションによる方法は、個々のアクション（イベント）に対して公平性を記述できる。プルダウン方式では、プロセスレベルの弱公平性、プロセスレベルの局所的強公平性、大域的強公平性、の３通りから選択できる。アクション（イベント）レベルの公平性とは、イベント毎に公平性を仮定することである。プロセスレベルの公平性とは、より細かな、一つ一つの遷移に対して公平性を仮定することを意味する。本稿の実験で利用したMaudeモデル検査器では、記述の方法によって、イベントレベルで公平性を記述することも、さらに細かなプロセスレベルで公平性を記述することも可能であった。Maudeでは、遷移は規則を用いて記述しているが、規則の一部を対象に仮定した場合は局所的公平性、全部を対象に仮定した場合は大域的公平性と表現するものである。PATは、Maudeで実例を示したような弱公平性と強公平性を混在してモデル検査することはできない。

## 5.2 The Maude Fair LTLR Model Checker

The Maude Fair LTLR Model Checkerは、Kyungmin Baeにより研究、開発[7]されており、既存のMaudeを拡張して利用することができる。状態ベースとイベントベースの両方を、モデル検査器の理論として採用している。理由は、状態ベースのみでも、イベントベースのみでも、表現力が不足するため[7]としている。本稿の実験で扱ったMaudeも、2.5節、2.6節で触れたように状態機械であるクリプケ構造に、イベントを追加したラベル付きクリプケ構造とSE-LTL式を用いることで、公平性を仮定したモデル検査を可能としていた。The Maude Fair LTLR Model Checkerは、既存のMaudeの機能拡張であるため、幾種もの並行・分散システムが記述された実績のある仕様記述言語であるMaude LTLモデル検査器をそのまま利用できる。このため、規則レベル、すなわちプロセスレベルの公平性の仮定が可能であり、また弱公平性と強公平性が混在するようなモデル検査も可能である。既存のMaudeでは、検査したい性質に対し、公平性の仮定を自由に表現することができるが、LTL式の肥大化が原因による時間的、空間的な検査時間の増大が問題となっていた。The Maude Fair LTLR Model Checkerはこの問題に対して、公平性の仮定のアルゴリズムを用いることで解決を試みている。アルゴリズムの実装も、効率化のためC++言語を採用している。加えて、The Maude Fair LTLR Model Checkerは既存のMaudeコードをサポートするため、本稿で調査、実験してきたDCAも併用できることを強調しておきたい。[7]では、いくつかの研究事例に対し、実際にThe Maude Fair LTLR Model Checkerを動作させた結果が報告されている。一定のサイズの問題で、既存のMaudeでは状態爆発が発生し解が得られなかったことに対し、The Maude Fair LTLR Model Checkerでは結果が得られたことを報告している。

本稿では、The Maude Fair LTLR Model Checkerを用いて、3.3節のQlock（プロセス２個、全ての遷移に対し公平性を仮定）の無排斥性をモデル検査し観察した。実験には、[13]からThe Maude Fair LTLR Model Checkerの実行ファイルである「Linux on Intel x86 and MacOSX on Intel」をダウンロードして使用した。Qlockに対し、既存のMaudeのLTL検査用ファイルmodel-checkerの読み込みをコメントアウトし、代わりにltlr-interfaceを読み込み、modelCheckFair()を適用して検査した結果は以下の通りである。

```
==========================================
reduce in PROC2MC : modelCheckFair(init2, lofree2, fair({'
   dose}) ; fair({'want}) ; fair({'exit}) ; fair({'try})) .
rewrites: 215 in 0ms cpu (1ms real) (446058 rewrites/second)
result ModelCheckResult: counterexample(
{ queue: empty le: notran (pc[p1]: rs) pc[p2]: rs,{'want : '
   E \ notran ; 'I \ p1 ; 'Q \ empty} }
{ queue: (p1 | empty) le: want(p1) (pc[p1]: ws) pc[p2]: rs
   ,{'dose : 'E \ want(p1) ; 'I \ p2} },
```

```
{ queue: (p1 | empty) le: dose(p2) (pc[p1]: ws) pc[p2]: rs
   ,{'dose : 'E \ dose(p2) ; 'I \ p2} })
==========================================
reduce in PROC2MC : modelCheckFair(init2, lofree2, just({'
   dose}) ; just({'want}) ; just({'exit}) ; just({'try})) .
rewrites: 215 in 1ms cpu (2ms real) (113277 rewrites/second)
result ModelCheckResult: counterexample(
{ queue: empty le: notran (pc[p1]: rs) pc[p2]: rs,{'want : '
   E \ notran ; 'I \ p1 ; 'Q \ empty} }
{ queue: (p1 | empty) le: want(p1) (pc[p1]: ws) pc[p2]: rs
   ,{'dose : 'E \ want(p1) ; 'I \ p2} },
{ queue: (p1 | empty) le: dose(p2) (pc[p1]: ws) pc[p2]: rs
   ,{'dose : 'E \ dose(p2) ; 'I \ p2} })
```

Listing 5.1: modelCheckFair() を Qlock に適用した例

modelCheckFair() は、初期状態、性質、公平性の 3 つを引数にとる。今回は、全ての遷移に弱公平性（fair）を仮定したものと、強公平性（just）を仮定した 2 つの試験を実施した。反例を得ることなく正常に終了することを期待していたが、どちらの場合も、3 番目の状態から無限ループする反例が出力されてしまい、実験として失敗してしまった。この原因としては、The Maude Fair LTLR Model Checker の modelCheckerFair() の 3 番目の引数である公平性を正しく記述できなかったことが考えられる。

## 5.3 まとめ

本章では、公平性の仮定をサポートした主だったモデル検査器の研究について調査し、本稿で利用した Maude 検査器にも触れつつ 2 つの研究事例について報告した。5.1 節では PAT について、5.2 節では既存の Maude を拡張した The Maude Fair LTLR Model Checker について記載した。既存の Maude では、検査したい性質の LTL 式に対し、公平性の仮定を直接埋め込むことで実現した。この方法は、表現力が高い一方で、LTL 式の肥大化が原因による時間的、空間的な検査時間の増大が課題となっていた。これに対し、PAT も The Maude Fair LTLR Model Checker も、公平性の仮定のサポートのための専用のアルゴリズムを用意し解決を図っていた。PAT は、プルダウン方式のような簡単な方法と、アノテーションによる方法の 2 つの方法で公平性の仮定を実現していた。The Maude Fair LTLR Model Checker は、既存の Maude 資産を拡張して公平性の仮定をサポートした。本稿で調査した手法である DCA は、既存の Maude で利用することができるため、The Maude Fair LTLR Model Checker と併用できる。もう一つの Maude の利点としては、LTL 式に直接記述することから可能となる、弱/強公平性などさまざまな公平性を混在させた検査ができる点も付記しておく。主だったモデル検査器についての調査報告をまとめたものは、図 5.1 の表である。

| 手法 | 公平性を仮定したモデル検査の可否 | 弱公平性 | 強公平性 | 特徴など |
|---|---|---|---|---|
| 分割統治アプローチ（DCA） | △ *1 | ○ | ○ | *1 LTL式に直接埋め込むことで公平性を実現。<br>・Maude (v2.7.1) で性能面の課題を一定程度解決。<br>・弱/強公平性を混在させた検査が可能。<br>・機械的な分割手順や、効率的な手順はまだ発見されてない。 |
| SPIN | ○ | ○ | × | ・弱公平性のみ扱える。 |
| SAL<br>(Symbolic Analisis Laboratory) | × | × | × | ・公平性は扱えない。 |
| PAT(3.5) | ○ | ○ *2 | ○ *2 | *2 選択式のoptionを用いて実現。<br>・annotationを用いた記述も可能。<br>・公平性の仮定をアルゴリズムで実装。 |
| NuSMV v2.6 | ○ | ○ | ○ | ・キーワードを用いて公平性を実現。<br>・公平性の仮定をアルゴリズムで実装。 |
| The Maude Fair LTLR model checker | ○ | ○ | ○ | ・実績のある既存のMaudeを拡張、そのまま利用できる。<br>・公平性の仮定をアルゴリズムで実装。<br>・弱/強公平性を混在させた検査が可能。 |

図 5.1: モデル検査器の公平性対応の一覧（2021 年 1 月 13 日時点）

# 第6章　おわりに

本章では、6.1節で、この課題研究報告書の目標であるDCAの有効性について、実験、調査を通して得た結論を報告する。6.2節で、DCAの課題を考察し、その改善提案を述べる。

## 6.1　まとめ

本課題研究は、活性のモデル検査におけるDCAの有効性を実証した。1章では、モデル検査そのものと、活性のモデル検査における公平性の重要性、活性のモデル検査が抱える課題について説明した。2章では、Maudeモデル検査器を用い、実例を交えつつ公平性を仮定したモデル検査、ならびにDCAの理論的な説明をした。同2章から4章にかけて、3つの具体例を実験し、DCAの適用/非適用で、活性のモデル検査を実施し、比較した。結果は、3つの具体例すべてにおいて、DCAを適用した場合に、時間的、空間的な効率化を示した。5章では、活性のモデル検査が抱える課題について、別のアプローチからの研究事例を調査し、比較した。公平性をサポートするどのモデル検査器も、記述の表現力と、利用の簡便性とにトレードオフをもち、効率化に対して研究し、改善を試みていた。DCAは、モデル検査器そのものではなく、記述する性質の時相論理式に対して働きかけるため、既存のモデル検査器に対しても広く適用できるものであった。以上より、DCAは、その時間的、空間的な効率化と、広く適用できる汎用性において、有効であると結論づけられる。

## 6.2　今後の課題

本節では、活性のモデル検査におけるDCAの、改善点について考察する。DCAは、既存のモデル検査器に対し適用できる手法であることが、その特徴の一つと言える。モデル検査器そのものの効率化の研究とは異なり、その適用範囲は広い。

一方、5章で調査した関連研究では、専用のモデル検査器を改善することで効率化を図っていた。これらのモデル検査器は、選択式やコマンド指定など簡易的な方法で、公平性の仮定を実現していた。これらは、表現力に制限がある（例えば弱/強公平性が混在したモデル検査ができない等）ものの、簡易的に利用できるという点で利用者の間口を広げるものと言える。

本稿で実験してきたMaude LTLモデル検査器は高い表現力を有するが、時相論理式のBüchiオートマトンへの変換による非効率は避けて通れない。DCAが、この非効率に有効であることは前節で述べた通りである。しかし、DCAを適用するためには、検査対象の性質である時相論理式を、一定の条件に合うように、利用者自身により手動で分割する必要がある。この方法は簡単ではなく、利用者の経験に大きく依存してしまう。

これを解消するための方法の一つとして、「DCAの機械的な方法論の確立」を提案する。DCA自体の正しさは、すでに [3] で証明されている。しかしながら、どのように分割すれば効率的であるかや、機械的に分割することができるかについては、まだ提示されていない。

機械的な分割の一つの方法として、総当たりによる分割が考えられる。2.8節の末尾で、一部の遷移に弱公平性、一部の遷移に強公平性、その他の遷移には公平性を仮定しないモデル検査の例を実験した。このモデル検査は反例を得ず、正しく検査が終わる例である。この例では、公平性の仮定の仕方を見つけるにあたり、計算機を用いて総当たりで出力するという方法を採用した。具体的には、全ての遷移に対し、仮定する/しない、を場合わけしたMaudeコードを総当たりで出力し、これらに対して実際にモデル検査して真偽を確かめた。ここで得られる論理式は、DCAに適合する一つの解にもなっている。この方法は小さな事例に対しては有効であるが、遷移を単位とするため、従来のモデル検査の問題と同様、容易に状態爆発してしまう。ゆえに、総当たりによる分割を方法として用いるためには、3.4節のQlockのDCA適用例で見たような、遷移を単位としないケースである必要がある。したがって、性質の時相論理式に対して、

- 総当たりで、遷移が減少する方向で機械的に合成し、
- 総当たりで、公平性の仮定をする/しない、を場合わけし、
- 総当たりで、DCAに合致する/しないを選別し、

その出力をもって利用者の分割判断の助けとする方法、を提案する。方法論が確立すれば、一定程度の自動化が可能となる。繰り返しになるが、DCAは適用範囲も広く、表現力も高いため、方法論の確立に関する研究は、有用であると言える。

# 付 録A　ソースコード

本稿の実験に用いたMaudeコードは、https://github.com/koyanagisatoshi/maude に登録している。

# 謝辞

本課題研究報告書の執筆にあたり、主指導教員である緒方和博教授には、数々の貴重なご指導、ご助言、並びに多くのお時間を賜りました。研究の着想から、調査の方法、報告書執筆の細部に至るまで、粘り強く、丁寧に、かつ熱意をもって導いていただきました。緒方先生のお力添えなしには、本稿は存在しません。深く、心より感謝申し上げます。平石邦彦教授、青木利晃教授、石井大輔准教授には、審査において貴重なお時間を割いていただき、限られた時間の中で多くのご指摘、ご助言を賜りました。本稿を改定するにあたり、深く、御礼を申し上げます。また、私生活を支えてくれた妻 妃奈子にも、深く感謝します。

# 索 引


always 演算子，14
assoc キーワード，10

comm キーワード，10
ctor キーワード，6

DCA（分割統治アプローチ），24

EES-KS(LKS の EES-KS)，18
eq キーワード，12，33，38，51
eventually 演算子，14

fmod キーワード，6

juxtaposition 演算子，10

leadsto 演算子，14
LTL 式，13

MODEL-CHECKER モジュール，14
modelCheck()，14，34

next 演算子，13

op、ops キーワード，6

pr キーワード，7

red コマンド，8
rl キーワード，11，32

SE-LTL 式，16
sort、sorts キーワード，6

until 演算子，13

var、vars キーワード，7

安全性，14，28

演算，6

活性，15，33
観測可能成分（= 名前と値の対），9

擬公平性，25
弱公平性，21
強公平性，21

クリプケ構造，11，30

結合法則，9

交換法則，9
公平性，21，34

サブソート，31

時相演算子，13，14
実行系列，13，16
状態，9
状態機械，8

スープ（= コレクション），9

相互排除，28
ソート，6

中置二項演算子，7

等式，7

パス，13，16

待ち行列，4

無排斥性，33

ラベル付きクリプケ構造，16，35
ランク，6

# 関連図書


[1] Edmund Clarke - A.M. Turing Award Laureate. https://amturing.acm.org/award_winners/clarke_1167964.cfm.

[2] 中島震. SPIN モデル検査: 検証モデリング技法. 近代科学社, 2008.

[3] Kazuhiro Ogata. A divide & conquer approach to liveness model checking under fairness & anti-fairness assumptions. *Frontiers Comput. Sci.*, Vol. 13, No. 1, pp. 51-72, 2019.

[4] Kazuhiro Ogata. Model checking liveness properties under fairness & anti-fairness assumptions. In Pornsiri Muenchaisri and Gregg Rothermel, editors, *20th Asia-Pacific Software Engineering Conference, APSEC 2013, Ratchathewi, Bangkok, Thailand, December 2-5, 2013 - Volume 1*, pp. 565-570. IEEE Computer Society, 2013.

[5] Kazuhiro Ogata and Min Zhang. A divide and conquer approach to model checking of liveness properties. In *37th Annual IEEE Computer Software and Applications Conference, COMPSAC 2013, Kyoto, Japan, July 22-26, 2013*, pp. 648-657. IEEE Computer Society, 2013.

[6] Yuanjie Si, Jun Sun, Yang Liu, Jin Song Dong, Jun Pang, Shao Jie Zhang, and Xiaohu Yang. Model checking with fairness assumptions using PAT. *Frontiers Comput. Sci.*, Vol. 8, No. 1, pp. 1-16, 2014.

[7] Kyungmin Bae and José Meseguer. Model checking linear temporal logic of rewriting formulas under localized fairness. *Sci. Comput. Program.*, Vol. 99, pp. 193-234, 2015.

[8] Fabio Somenzi and Roderick Bloem. Efficient büchi automata from LTL formulae. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000,*

*Proceedings*, Vol. 1855 of *Lecture Notes in Computer Science*, pp. 248-263. Springer, 2000.

[9] Paul Gastin and Denis Oddoux. Fast LTL to büchi automata translation. In Gérard Berry, Hubert Comon, and Alain Finkel, editors, *Computer Aided Verification, 13th International Conference, CAV 2001, Paris, France, July 18-22, 2001, Proceedings*, Vol. 2102 of *Lecture Notes in Computer Science*, pp. 53-65. Springer, 2001.

[10] 緒方和博，中村正樹，二木厚吉．Maude：書換え論理に基づく計算機言語および処理系．コンピュータ ソフトウェア，Vol. 25，No. 2，2008.

[11] Maude Manual (Version 2.7.1). http://maude.cs.illinois.edu/w/images/e/e0/Maude-2.7.1-manual.pdf.

[12] Process Analysis Toolkit (PAT) 3.5 User Manual. https://www.comp.nus.edu.sg/~pat/OnlineHelp/index.htm.

[13] The Maude Linear Temporal Logic of Rewriting Model Checker. http://maude.cs.illinois.edu/tools/tlr/.