JAIST Repository
https://dspace.jaist.ac.jp/

Title	【課題研究報告書】Maudeを用いた形式仕様作成とモデル検査の調査研究
Author(s)	中村, 剛
Citation	
Issue Date	2024-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/18917
Rights	
Description	Supervisor: 緒方 和博, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

課題研究報告書

Maude を用いた形式仕様作成とモデル検査の調査研究

中村 剛

主指導教員 緒方 和博 教授

北陸先端科学技術大学院大学
先端科学技術研究科
（情報科学）

令和 6 年 3 月

概要

近年あらゆるものがデジタル化されている。さらに技術革新が加速し、AI や IoT などの新しい技術が登場しシステムの重要性はますます高まっている。こうした中で継続的にシステムがアップデートを繰り返しながら進化していく為、システムが安全に稼働する事はとても重要である。その為の一つの手法としてモデル検査がある。このモデル検査を正しく扱えるように本課題研究報告書では、いくつかの事例を用いて形式仕様の作成とモデル検査についてまとめる。すでに知られている事例を題材にするが、あえて誤りを注入したものを取り扱い理解を深めた。調査研究を進める中で遭遇した課題（状態爆発)、現象（状態が無限に作り出される現象）についても報告書でまとめ、対策を施しながらまとめている。

本課題研究報告書は 7 章から構成される。第 1 章では本課題研究報告書の背景、目的を述べて続く章の構成に関してまとめている。第 2 章では状態機械を定義して不可分な命令である test&set を利用した相互排除プロトコルを用いてどのように状態を表現するかについて述べ、Maude での記述方法、Search コマンドによるモデル検査方法についてまとめた。第 3 章では不可分な待ち行列を利用した相互排除プロトコルである QLock を用いて形式仕様の作成、Maude での記述、モデル検査についてまとめた。Qlock では相互排除性を満たさない Qlock を扱いながら違いについてもまとめた。第 4 章では敵味方を識別する為のプロトコルである Identity-Friend-or-Foe 認証プロトコル (IFF) を用いて形式仕様の作成、Maude での記述、モデル検査についてまとめた。ここでは本学の大容量メモリ計算機である Large Memory PC Cluster (LMPCC) を活用したモデル検査の実施や状態爆発の課題に直面しながら改善した内容をまとめている。第 5 章と 6 章では公開鍵暗号を用いた相互認証プロトコルの Needham-Schroeder 公開鍵認証プロトコル (NSPK)、NSPK の改良版である Needham ‐ Schroeder ‐ Lowe 公開鍵認証プロトコル (NSLPK) を用いて前の章で得られた知見をもとに形式仕様の作成、Maude での記述、モデル検査についてまとめた。第 7 章では課題研究報告書を通して得られた知見をまとめると共に近い将来直面する課題と対策を考察し、言及する。

目 次

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

第2章 準備 **3**
- 2.1 状態機械 . . . 3
- 2.2 Maude . . . 6
  - 2.2.1 test&set を用いる相互排除プロトコルの Maude による記述（形式仕様作成） . . . 6
  - 2.2.2 test&set を用いる相互排除プロトコルの Maude による不変性のモデル検査 . . . 8
- 2.3 Flawed Test & Set . . . 9

第3章 相互排除プロトコル Qlock **11**
- 3.1 相互排除性を満たさない Qlock(FQlock) . . . 11
  - 3.1.1 疑似コード . . . 11
  - 3.1.2 図解 . . . 12
  - 3.1.3 観測可能成分 . . . 15
  - 3.1.4 状態機械としての形式化 . . . 16
  - 3.1.5 Maude による形式仕様 . . . 17
  - 3.1.6 Maude の search コマンドによる不変性モデル検査 . . . 19
  - 3.1.7 Queue の値が積み上がる構造 . . . 20
- 3.2 FQlock の修正 . . . 24
  - 3.2.1 疑似コード . . . 24
  - 3.2.2 図解 . . . 25
  - 3.2.3 状態機械としての形式化 . . . 28
  - 3.2.4 Maude による形式仕様 . . . 29
  - 3.2.5 Maude の search コマンドによる不変性モデル検査 . . . 29
- 3.3 プロセス数を増やした場合の Qlock . . . 30
  - 3.3.1 実験結果 . . . 31
  - 3.3.2 既存研究 . . . 32

第4章 Identity-Friend-or-Foe 認証プロトコル (IFF) 33
4.1 観測可能成分 34
4.2 Maude におけるメッセージ送信/返信の表現 34
4.3 敵味方の識別性を満たさない IFF 34
4.3.1 状態機械としての形式化 35
4.3.2 Maude による形式仕様 37
4.3.3 Maude の search コマンドによる不変性モデル検査 39
4.4 FIFF の修正 39
4.4.1 Maude の search コマンドによる不変性モデル検査 40
4.5 状態爆発対策 40
4.5.1 改良した Maude のコード 40
4.5.2 LMPCC での実行結果 41
4.5.3 さらに改良した Maude のコード 41
4.6 別の状態爆発対策 42
4.6.1 Maude の search コマンドによるモデル検査 43
4.7 状態爆発対策の実験のまとめ 44
4.8 状態爆発対策のまとめ 44

第5章 Needham-Schroeder 公開鍵認証プロトコル (NSPK) 45
5.1 観測可能成分 46
5.2 状態機械としての形式化 47
5.3 Maude による形式仕様 48
5.4 Maude の search コマンドによる不変性モデル検査 50

第6章 Needham - Schroeder - Lowe 公開鍵認証プロトコル (NSLPK) 52
6.1 Maude による形式仕様 52
6.2 Maude の search コマンドによる不変性モデル検査 52
6.3 状態爆発対策 53
6.4 Maude の search コマンドによる不変性モデル検査 55

第7章 おわりに 56
7.1 まとめ 56
7.2 今後の課題 58
7.2.1 Large-Scale Directed Model Checking LTL 59
7.2.2 A Divide & Conquer Approach to Leads-to Model Checking 59

図目次

2.1 TASの図解 4
2.2 同時にcsに入るまでの遷移 10

3.1 初期状態 12
3.2 p1が図3.1から状態遷移1を行った状態 13
3.3 p2が図3.2から状態遷移1を行った状態 13
3.4 p1が図3.3から状態遷移2を行った状態 14
3.5 p1が図3.4から状態遷移3を行った状態 14
3.6 p2が図3.5から状態遷移2を行った状態 15
3.7 p2が図3.6から状態遷移3を行った状態 15
3.8 state 33までの経路 20
3.9 Queue変数が積み上がる遷移 21
3.10 共有変数 *queue* が積み上がる 22
3.11 初期状態 25
3.12 p1が図3.11から状態遷移1を行った状態 25
3.13 p2が図3.12から状態遷移1を行った状態 26
3.14 p1が図3.13から状態遷移2を行った状態 26
3.15 p1が図3.14から状態遷移3を行った状態 27
3.16 p2が図3.15から状態遷移2を行った状態 27
3.17 p2が図3.16から状態遷移3を行った状態 28
3.18 初期状態から始まる全ての到達可能状態 30

4.1 IFFメッセージ交換 33
4.2 誤認が成立する概要図 35
4.3 状態爆発対策を考慮したIFFのメッセージ交換 42

5.1 NSPKのメッセージ交換 45
5.2 なりすましが成立したメッセージ交換 51

表 目 次

4.1　変数 P と変数 Q に入る組み合わせ 43

第1章　はじめに

1.1　背景

近年ではあらゆるものがデジタル化され、ソフトウェアやシステムの安全性、信頼性に対する要求は高くなっている。またソフトウェアやシステムは複雑化、大規模化が進んでおり従来のようなレビューやテストでは全てのケースを検証する事は困難になっている。このような中でも継続的にシステムがアップデートを繰り返しながら進化していく為、安全に稼働する事は重要である。モデル検査はソフトウェアやシステムがもつ性質をモデルで表現し、すべての到達可能状態を網羅的に探索することで不具合や誤りを検出する。網羅的に探索する為、普段では気づくことができない特別な状態も発見可能である。気づきにくい不具合のケースも検出できる為、品質の高いソフトウェア、システムを開発する為の有効な技術のひとつである。

1.2　目的

本課題研究報告書の目的はMaudeでシステムの形式仕様を作成する方法とシステムが不変性を満たすことのモデル検査方法の調査である。その為、いくつかの事例を用いて形式仕様の作成、Maudeでの記述、不変性のモデル検査の実施について調査する。また意図的に誤りを注入した事例も扱うことで反例の抽出、反例が発生するまでの遷移の調査、モデル検査における課題のひとつである状態爆発問題が発生するまでと対策方法についても理解を深める。また状態が無限に積み上がり状態爆発問題につながる現象に対しても理解を深める。近い将来直面する課題や今回の課題研究報告書では扱っていない性質の検査についても概要をまとめる。今後モデル検査をより活用する上では重要になる為、今後の学修につなげていく。

1.3　構成

残りの章の構成を以下に記す。

- 第 2 章では状態機械を定義し、test&set を利用した相互排除プロトコルを用いて形式仕様の作成、Maude での記述、不変性のモデル検査について述べる。また意図的に誤りを埋め込んだ test&set を用いて不変性のモデル検査の動作について述べる。
- 第 3 章では不可分な待ち行列を利用した相互排除プロトコルである QLock を用いて形式仕様の作成、Maude での記述、不変性のモデル検査について述べる。またここでも意図的に誤りを埋め込んだ相互排除性を満たさない Qlock(FQlock) を用いて形式仕様の作成、Maude での記述、不変性のモデル検査について述べる。到達可能状態が無限に作られる構造についても述べる。
- 第 4 章では敵味方を識別する為のプロトコルである Identity-Friend-or-Foe 認証プロトコル (IFF) を用いて形式仕様の作成、Maude での記述、モデル検査について述べる。ここでは状態爆発の課題についても述べる。
- 第 5 章では Needham-Schroeder 公開鍵認証プロトコル (NSPK) を用いて形式仕様の作成、Maude での記述、モデル検査について述べる。
- 第 6 章では NSPK の改良版である Needham - Schroeder - Lowe 公開鍵認証プロトコル (NSLPK) を用いて形式仕様の作成、Maude での記述、モデル検査について述べる。
- 第 7 章では課題研究報告書を通して得られた知見をまとめ、今後の課題に関して述べる。

第2章　準備

2.1　状態機械

状態機械は、相互排除プロトコルや認証プロトコルなどさまざまなプロトコルやシステムを形式化可能である。状態機械は、状態の集合 S、初期状態の集合 $I \subseteq S$、それに状態集合 S 同士の直積集合の部分集合 $T \subseteq S \times S$ から構成される。$M \triangleq \langle S, I, T \rangle$ と表現できる。T の要素 (s, s') は、状態 s から状態 s' への移り変わり（状態遷移）とみなすことができる。このため、(s, s') を状態遷移と呼び、T を状態遷移の集合と呼ぶ。

ここで以下の疑似コードで記される相互排除プロトコルを考える。

1 疑似コード Test & Set

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

3 つの領域 rs (reminder section)、 ws (waiting section)、 cs (critical section) をプロセスが移動していく事を想定する。*locked* のブール値はそれぞれのプロセスで共有する値になり、cs に入る際と出る際に確認と変更されるものである。プロセスが cs に入る際は test&set 機構を使い、高々一つを許容する。test&set は *locked* の値を必ず true に変更する。引数で渡された *locked* の値が false であれば、false を返却し、true であれば true を返却する。その事により、*locked* の値の確認と変更を不可分に行う事ができる。test&set を用いる相互排除プロトコルのことを TAS と呼ぶ。次にプロセスに着目し、各領域での動作について説明する。

初期状態では各プロセスは rs にいる。*locked* の値は false である。rs に留まっている期間を経て、次にプロセスは ws に移動する。ws では *locked* が false であれば、cs に移動し、*locked* を true に変更する。cs から rs へ戻る際は *locked* の値を false に変更する。

図解で表すと下記のようになる。

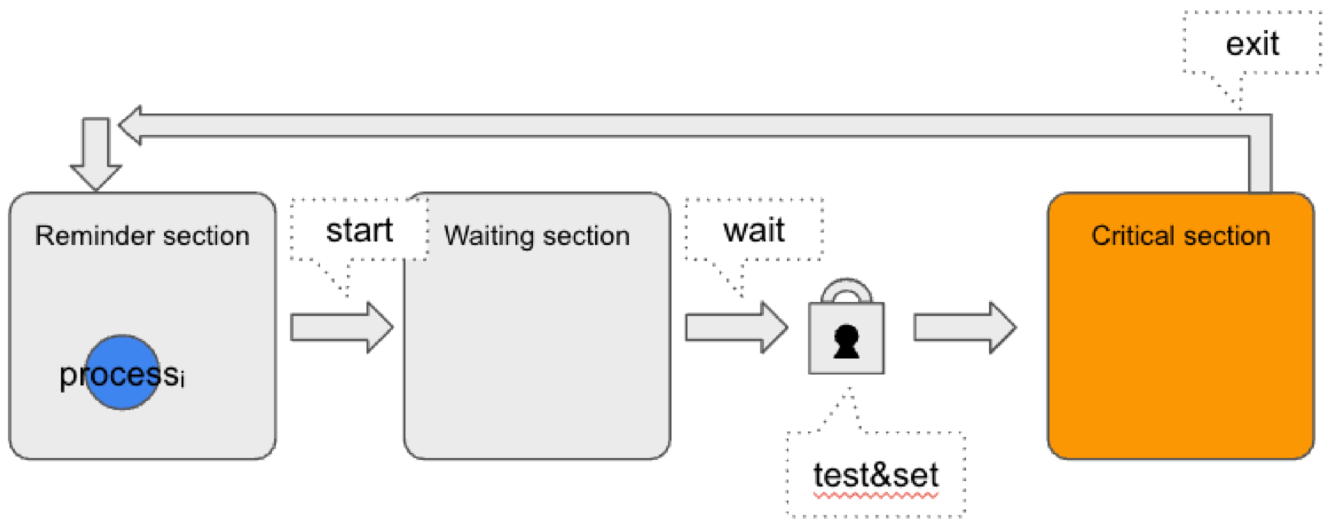


図 2.1: TAS の図解

状態の表現方法は行く通りもあるが、この課題研究報告書では、観測可能成分のスープを中カッコで囲ったもので表現する。観測可能成分は名前と値の組である。たとえば、ブール値の変数 *locked* を観測可能成分で書き表すと (locked: b) となる。ここで、locked は名前で、b は値（true あるいは false）である。別の例として、あるプロセス p は 3 つの位置 rs,ws,cs のいずれかに存在するとすれば、このことを観測可能成分で書き表すと (pc[p]: l_p) となる。ここで、pc[p] は名前で、l_p は値（位置 rs,ws,cs のいずれか）である。プロセス p に加え、別のプロセス q もいて、同様に 3 つの位置のいずれかに存在し、ブール値の変数 *locked* を共有しているとする。このようなシステム・プロトコルの状態は、{(locked: b) (pc[p]: l_p) (pc[q]: l_q)} で表現することができる。{(locked: b) (pc[p]: l_p)(pc[q]: l_q)} は、順番を入れ替えても等しさは変わらない。このため、以下はすべて同じ状態を表す。

- {(locked: b) (pc[p]: l_p) (pc[q]: l_q)}
- {(locked: b) (pc[q]: l_q) (pc[p]: l_p)}
- {(pc[p]: l_p) (locked: b) (pc[q]: l_q)}
- {(pc[p]: l_p) (pc[q]: l_q) (locked: b) }
- {(pc[q]: l_q) (locked: b) (pc[p]: l_p)}
- {(pc[q]: l_q) (pc[p]: l_p) (locked: b)}

状態の集合 S は下記のように定義する。

$$S \triangleq \{ \{(\mathrm{pc[p]}: l_p)\ (\mathrm{pc[q]}: l_q)\ (\mathrm{locked}: B)\} \mid l_p, l_q \in \mathrm{Loc},\ B \in \mathrm{Bool} \}$$

Loc はロケーション識別子の集合 (ソート) である。Bool はブール値の集合 (ソート) である。

初期状態の集合 I は下記のように定義する

$$I \triangleq \{\{(\mathrm{pc[p]}: \mathrm{rs})\ (\mathrm{pc[q]}: \mathrm{rs})\ (\mathrm{locked}: \mathrm{false})\}\}$$

状態から状態への遷移の T については下記のように定義する。

$$\begin{aligned}
T \triangleq \{(& \\
&\{(\mathrm{pc}[P]: \mathrm{rs})\ (\mathrm{locked}: B)\ OCs\},\\
&\{(\mathrm{pc}[P]: \mathrm{ms})\ (\mathrm{locked}: B)\ OCs\}\\
&) \mid P \in \mathrm{Pid},\ B \in \mathrm{Bool},\ OCs \in \mathrm{OComp}\\
\} \cup \{(& \\
&\{(\mathrm{pc}[P]: \mathrm{ws})\ (\mathrm{locked}: \mathrm{false})\ OCs\},\\
&\{(\mathrm{pc}[P]: \mathrm{cs})\ (\mathrm{locked}: \mathrm{true})\ OCs\}\\
&) \mid P \in \mathrm{Pid},\ B \in \mathrm{Bool},\ OCs \in \mathrm{OComp}\\
\} \cup \{(& \\
&\{(\mathrm{pc}[P]: \mathrm{cs})\ (\mathrm{locked}: \mathrm{true})\ OCs\},\\
&\{(\mathrm{pc}[P]: \mathrm{rs})\ (\mathrm{locked}: \mathrm{false})\ OCs\}\\
&) \mid P \in \mathrm{Pid},\ B \in \mathrm{Bool},\ OCs \in \mathrm{OComp}\\
\}&
\end{aligned}$$

Pid はプロセス識別子の集合（ソート）である。OComp は観測可能成分の集合 (ソート) である。

次に到達可能状態について説明する。状態機械 M の到達可能状態の集合 R は下記のように定義する。

1. $I \subseteq R$
2. $(s,s') \in T$ かつ $s \in R$ であれば $s' \in R$

初期状態は集合の部分集合であり、初期状態からスタートする為、到達可能状態である。ある状態 s と遷移でつながれた s の一つ先の状態 s' があった時、s が到達可能であれば、s から遷移でつながれた s' も到達可能である。

次に状態述語について説明する。状態述語とは $p : S \rightarrow \mathrm{Bool}$ である述語である。状態述語は全称限定子で束縛されている変数を含むこともある。$p(s) \triangleq (\forall\ p, q : \mathrm{Pid})\ p(s,p,q)$ ここでの s は S の変数である。全てのプロセスにおいて、Bool 値で表せる述語である。

次に不変性について説明する。$(\forall\ s : R)\ p(s)$ を満たす状態述語のことであり、すべての到達可能状態で成り立つ状態述語のことである。

2.2 Maude

Maude[1] について説明する。Maude は書換え論理に基づく形式仕様言語またはプログラミング言語である。アルゴリズムや検証対象のシステムの振る舞いを記述することが可能であり、網羅的な検査が可能である。アルゴリズムやシステムが満たすべき性質が保たれているかを網羅的に検査する。網羅的に検査を実行する為、普段では気づけないようなレアケースの不具合の発見が可能になる。検査式が成り立たなかった場合には反例が出力される。反例は初期状態から不具合になる状態までの状態遷移ログが出力され、追って解析が可能になる。

2.2.1 test&set を用いる相互排除プロトコルの Maude による記述（形式仕様作成）

プロセスが移動する 3 つの領域とプロセスを下記のように定義する

```
1 fmod LOC is
2   sort Loc .
3   ops rs ws cs : -> Loc [ctor] .
4 endfm
5
6 fmod PID is
7   sort Pid .
8   ops p1 p2 p3 : -> Pid [ctor] .
9 endfm
```

モジュールは fmod から endfm までになり、LOC (location)、PID (process id) モジュールを定義している。最初にソート (sort) を定義しており、これはデータの型の意味になる。op は演算であり、ops は op の複数形になり、複数の演算が同時に宣言されている。: の左側は演算の名前を記述し、右側に具体的な定義を記述する。-> の左側は入力のソート、右側を出力のソートを記述する。rs,ws,cs の入力引数のソートは空の為、定数になる。[と] で囲われているものは属性を意味する。ctor は constructor であり、他にも assoc (結合法則)、comm (交換法則) などがある。

次に OCOMP モジュールを使い、各プロセスがどの Location にいて *locked* の値がどのような状態になっているかを表す。

```
1 fmod OCOMP is
2   pr PID .
3   pr LOC .
4   sort OComp .
5   op (pc[_]:_) :  Pid Loc -> OComp [ctor] .
6   op (locked:_) : Bool -> OComp [ctor] .
7 endfm
```

pr で先に定義したモジュールをインポートして活用する。このモジュールでは演算子として `(pc[_]:_)` と `(locked:_)` を定義して、OComp ソートへ変換している。OComp は Observable Component の略で観測可能成分である。

次にパラメータ付きモジュールについて説明する。

```
view OComp from TRIV to OCOMP is
  sort Elt to OComp .
endv
```

このビューは、モジュール TRIV のソート Elt を、モジュール OCOMP のソート OComp に置き換える事を意味している。

```
fmod SOUP {D :: TRIV} is
  sort Soup{D} .
  subsort D$Elt < Soup{D} .
  op empty : -> Soup{D} [ctor] .
  op _ _ : Soup{D} Soup{D} -> Soup{D} [ctor assoc comm id: empty] .
endfm
```

パラメータ化されたモジュールは、パラメータのソートや演算を実際のパラメータモジュールのソートや演算に置き換えてインスタンス化する。よって汎用的なモジュールとして扱うことが出来る。このモジュールでは結合法則、交換法則を満たした状態の集合をスープで定義している。

次に初期状態の集合を Maude のコードで記載したのが下記である。

```
fmod CONFIG is
  pr SOUP{OComp} .
  sort Config .
  op {_} : Soup{OComp} -> Config [ctor] .
  op init : -> Config .
  eq init = {(pc[p1]: rs) (pc[p2]: rs) (locked: false)} .
endfm
```

２つのプロセス p1 と p2 が存在するとする。初期状態ではともに rs に存在し、*locked* は false である。初期状態を init で参照できるようにしている。

次に状態から状態への遷移を表す書き換え規則を記したのもが下記である。

```
mod TAS is
  inc CONFIG .
  var B : Bool .
  var I : Pid .
  vars L1 L2 : Loc .
  var OCs : Soup{OComp} .
  rl [start] : {(pc[I]: rs) OCs} => {(pc[I]: ws) OCs} .
  rl [wait] : {(pc[I]: ws) (locked: false) OCs} => {(pc[I]: cs) (locked: true)
      OCs} .
  rl [exit] : {(pc[I]: cs) (locked: B) OCs} => {(pc[I]: rs) (locked: false) OCs
      } .
endm
```

rl が rewrite rule の意味になり、rl [name] : s => s' という書式になる。name は rewrite rule のラベルであり、状態 s から状態 s' への遷移を意味する。OCs はその他のスープを表している。それぞれの遷移は下記の通りである。

- start はプロセスを reminder section から waiting section に遷移

- wait はプロセスを waiting section から *locked* の値が false ならば critical section に遷移
- exit はプロセスが critical section から reminder section へ遷移し、lock を false に変換

2.2.2 test&set を用いる相互排除プロトコルの Maude による不変性のモデル検査

Maude による不変性のモデル検査は下記の書式になる。

```
search [n] in Module : s =>* s' .
```

n は検出したい状態の数の設定である。s は状態であり、s′ は検索対象の状態である。 =>は左の状態 s から右の状態 s′ に書き換える事を示す。 * は書き換えステップが 0 以上を示す。今回検査した Search コマンドは下記である。

```
1 search [1] in TAS : init =>* {(pc[p1]: cs) (pc[p2]: cs) OCs} .
```

init(初期) の状態から {(pc[p1]: cs) (pc[p2]: cs) OCs} にパターンマッチする状態を探索して 1 件でも存在すれば結果（ソルーション）を１つ返す。結果は下記のように 0 件で存在しない為、No solution となる。到達可能状態において p1 と p2 が同時に cs に存在する状態は 1 つも存在しなかった事になる。

```
1 Maude> search [1] in TAS : init =>* {(pc[p1]: cs) (pc[p2]: cs) OCs} .
2 search [1] in TAS : init =>* {OCs (pc[p1]: cs) pc[p2]: cs} .
3
4 No solution.
5 states: 8  rewrites: 15 in 0ms cpu (0ms real) (99337 rewrites/second)
```

2.3 Flawed Test & Set

test&set を用いる相互排除プロトコルの事を TAS と呼び。TAS に誤りのある（相互排除性を満たさない）ものを FTAS(Flawed Test & Set) と呼ぶ。誤りをわざと注入した FTAS を用いることで TAS の性質が満たされなった場合を作り出す事が可能になる。性質が満たされなかった場合の Maude の動作を理解する。ここでは FTAS を題材に Maude での記述方法、Search コマンドでの不変性のモデル検査を説明する。ここで以下の疑似コードで記される相互排除性を満たさない TAS を考える。

2 疑似コード Flawed Test & Set

1: **Loop**:
2: rs: ”Remainder Section”
3: ws: **repeat while** *locked*;
4: ss: *locked* := true;
5: ”Critical Section”
6: cs: *locked* := false;

TAS との違いは *locked* の値を false に変更するとともに元の値を返すということを不可分（アトミック）に行うか不可分に行わないかであり、FTAS は不可分に行わない。そのために ss ラベルを追加している。

Maude での違いは下記の遷移を表す書き換え規則が異なる。

TAS の書き換えコード

```
1  rl [start] : {(pc[I]: rs) OCs} => {(pc[I]: ws) OCs} .
2  rl [wait] : {(pc[I]: ws) (locked: false) OCs} => {(pc[I]: cs) (locked: true)
       OCs} .
3  rl [exit] : {(pc[I]: cs) (locked: B) OCs} => {(pc[I]: rs) (locked: false) OCs
       } .
```

FTAS の書き換えコード

```
1  rl [start] : {(pc[I]: rs) OCs} => {(pc[I]: ws) OCs} .
2  rl [wait] : {(pc[I]: ws) (locked: false) OCs} => {(pc[I]: ss) (locked: false)
       OCs} .
3  rl [set] : {(pc[I]: ss) (locked: B) OCs} => {(pc[I]: cs) (locked: true) OCs}
       .
4  rl [exit] : {(pc[I]: cs) (locked: B) OCs} => {(pc[I]: rs) (locked: false) OCs
       } .
```

TAS の wait では ws から cs セクションに移動すると共に *locked* の値を false から true に変更している。FTAS の wait では ws から ss セクションへ移動し、*locked* の値は変更されれない。FTAS の set で ss から cs セクションへ移動し、*locked* の値を true に変更している。つまり TAS では critical section に入ると同時に *locked* の値変更を行っていたが、FTAS の場合では *locked* の値の確認と変更を不可分な処理

では行わず、間を開けて行っていることになる。そのことにより、critical section に 2 つ以上のプロセスが入ってしまう為、相互排除性を満たさない。

Maude による不変性のモデル検査を実施した結果が下記になる。

```
Maude> search [1] in TAS-FLAWED : init =>* {(pc[p1]: cs) (pc[p2]: cs) OCs} .
search [1] in TAS-FLAWED : init =>* {OCs (pc[p1]: cs) pc[p2]: cs} .

Solution 1 (state 15)
states: 16  rewrites: 26 in 0ms cpu (0ms real) (139037 rewrites/second)
OCs --> locked: true
```

結果はソリューションを 1 つ発見した。プロセス p1 と p2 が同時に critical section にいる状態が 1 つ発見できたという意味になる。この状態になるまでの初期状態からの遷移を調べるコマンドは show path コマンドで下記のように出力される。

```
Maude> show path 15 .
state 0, Config: {locked: false (pc[p1]: rs) pc[p2]: rs}
===[ rl {OCs pc[I]: rs} => {OCs pc[I]: ws} [label start] . ]===>
state 1, Config: {locked: false (pc[p1]: ws) pc[p2]: rs}
===[ rl {OCs pc[I]: rs} => {OCs pc[I]: ws} [label start] . ]===>
state 3, Config: {locked: false (pc[p1]: ws) pc[p2]: ws}
===[ rl {OCs locked: false pc[I]: ws} => {(pc[I]: ss) OCs locked: false} [label
     wait] . ]===>
state 6, Config: {locked: false (pc[p1]: ss) pc[p2]: ws}
===[ rl {OCs locked: false pc[I]: ws} => {(pc[I]: ss) OCs locked: false} [label
     wait] . ]===>
state 10, Config: {locked: false (pc[p1]: ss) pc[p2]: ss}
===[ rl {OCs locked: B pc[I]: ss} => {(pc[I]: cs) OCs locked: true} [label set]
     . ]===>
state 13, Config: {locked: true (pc[p1]: cs) pc[p2]: ss}
===[ rl {OCs locked: B pc[I]: ss} => {(pc[I]: cs) OCs locked: true} [label set]
     . ]===>
state 15, Config: {locked: true (pc[p1]: cs) pc[p2]: cs}
```

わかりやすく図にしたものは下記である。一番左にある状態が初期状態になり、矢印は遷移を表す。背景が赤い色はプロセス p1,p2 が同時に cs に入った事を示す。

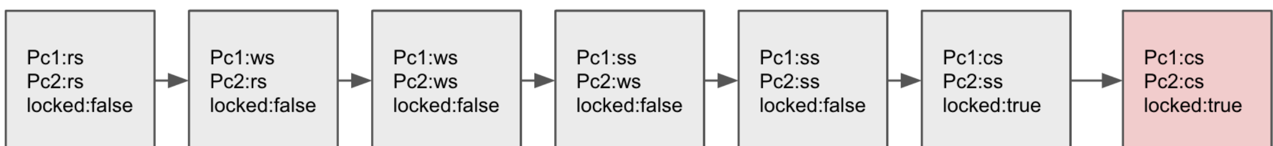


図 2.2: 同時に cs に入るまでの遷移

第3章　相互排除プロトコルQlock

複数のプロセスが同じ資源を同時に利用すると競合状態が生じる。あるプロセスが資源を利用している間は他のプロセスによる資源の利用を禁止する仕組みが存在する。その仕組みを相互排除プロトコルと呼び、不可分な待ち行列を利用する事で相互排除プロトコルを実現したものが Qlock である。ここでは Qlock と意図的に誤りを埋め込んだ相互排除プロトコル FQlock(Flawed Qlock) を題材にしてどのように状態機械として形式化を行ったか、どのように Maude で記述したか、モデル検査した保証すべき性質の内容、それに対する反例の説明を行う。

3.1　相互排除性を満たさないQlock(FQlock)

この FQlock では下記の誤りをあえて注入する。

- 待ち行列の共有変数 $queue$ の末尾にプロセスの識別子を追加する enqueue が不可分に行われていない
- 待ち行列の共有変数 $queue$ の先頭からプロセスの識別子を取り出す dequeue が不可分に行われていない

3.1.1　疑似コード

3 疑似コード FQlock

1: **Loop** {
2: "Remainder Section"
3: rs: tmp_i := enqueue($queue$,i)
4: es: $queue$:= tmp_i
5: ws: **repeat until** top($queue$)=i;
6: "Critical Section"
7: cs: tmp_i := dequeue($queue$);
8: ds: $queue$:= tmp_i;
9: }

疑似コードから下記の状態遷移が発生することが分かる。プロセス p1 と限定して記載しているが、実際にはプロセス p1 に限らず、FQlock に参加しているいかなるプロセスも対象となる

1. 状態遷移 1
   p1 が rs にいれば es に移動すると共に $queue$ の保持するプロセス識別子の待ち行列の末尾に p1 の識別子を追加しそれにより得られる新しいプロセス識別子の待ち行列を p1 の局所変数 tmp_i に代入する

2. 状態遷移 2
   p1 が es にいれば ws に移動すると共に $queue$ に p1 の局所変数の値を代入する

3. 状態遷移 3
   p1 が ws で、$queue$ の値の先頭が p1 であれば cs に移動する

4. 状態遷移 4
   p1 が cs にいれば ds に移動すると共に $queue$ の値の先頭を削除して得られるプロセス識別子の待ち行列を p1 の局所変数 tmp_i に代入する

5. 状態遷移 5
   p1 が ds にいれば rs に移動すると共に p1 の局所変数の値を $queue$ に代入する

3.1.2 図解

ここでは疑似コードから p1 プロセス、p2 プロセスの 2 つが存在していた場合の遷移の一例を図で説明する。矢印はプロセスの移動方向を表す。p1 と p2 はプロセスを表す。tmp_1 は p1 プロセスの局所変数を表す。tmp_2 は p2 プロセスの局所変数を表す。$queue$ は共有変数を表す。

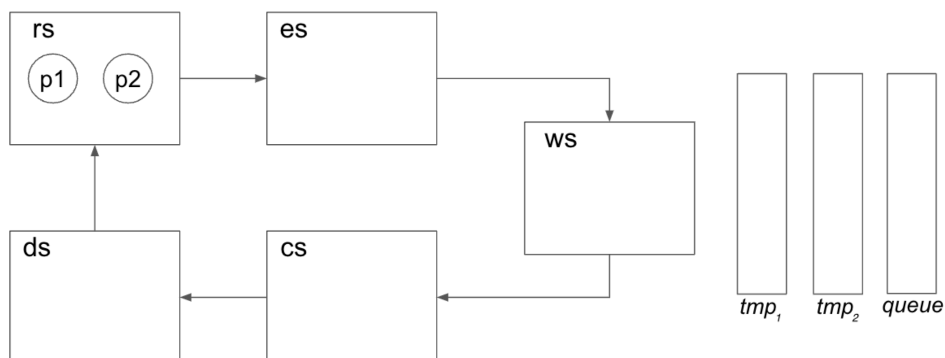


図 3.1: 初期状態

p1 と p2 プロセスは初期状態なので共に rs にいる。それぞれの局所変数、共有変数は空である。図 3.1 の初期状態からプロセス p1 が状態遷移 1 を行った場合は下記になる。

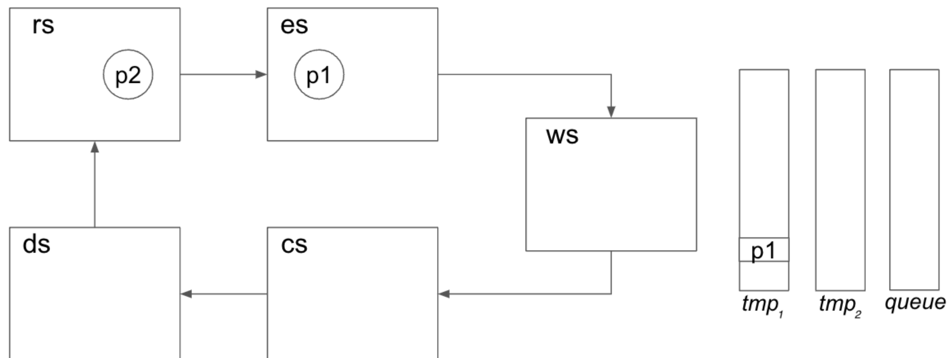


図 3.2: p1 が図 3.1 から状態遷移 1 を行った状態

p1 は es に移動したと同時に p1 の局所変数 tmp_1 も変化した。$queue$ が空だった為、空の待ち行列の末尾に p1 の識別子を追加し、tmp_1 に代入された。続いて図 3.2 からプロセス p2 も同様に状態遷移 1 を行うと下記の図になる。

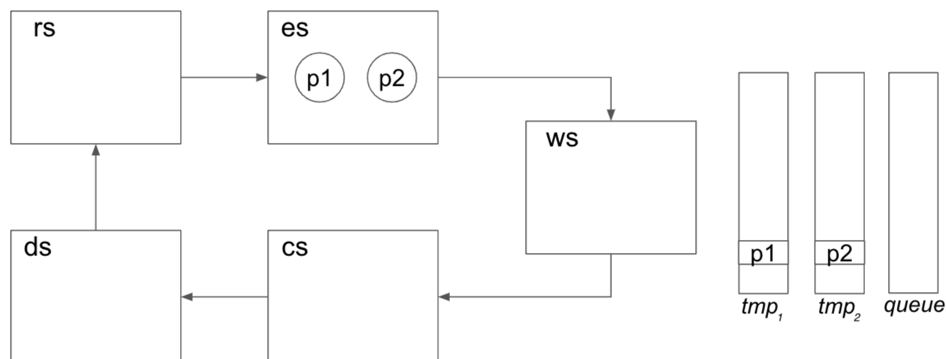


図 3.3: p2 が図 3.2 から状態遷移 1 を行った状態

p1 と同様に es に移動したと同時に p2 の局所変数 tmp_2 も変化した。同様に $queue$ が空だった為、空の待ち行列の末尾に p2 の識別子を追加し、tmp_2 に代入された。続いて図 3.3 から p2 が状態遷移 2 を行った場合は下記になる。

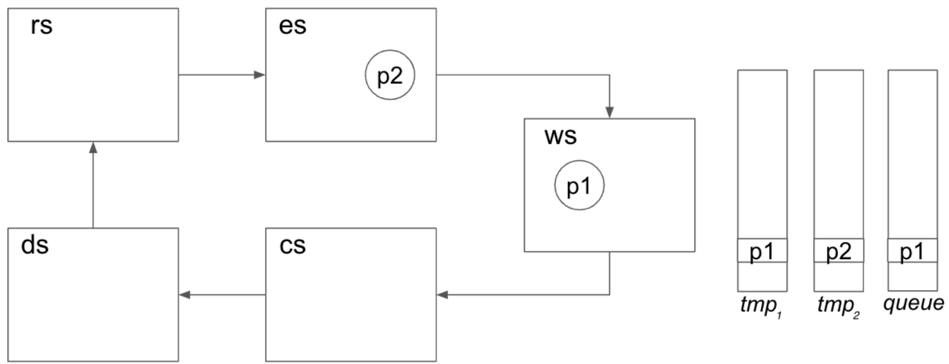


図 3.4: p1 が図 3.3 から状態遷移 2 を行った状態

p1 の局所変数 tmp_1 の待ち行列を共有変数 $queue$ へ代入した為 $queue$ は p1 となっている。続いて p1 は ws にいて、且つ共有変数 $queue$ の待ち行列の先頭が p1 である為、p1 が状態遷移 3 を行える。図 3.4 から p1 が状態遷移 3 を行った場合は下記になる。

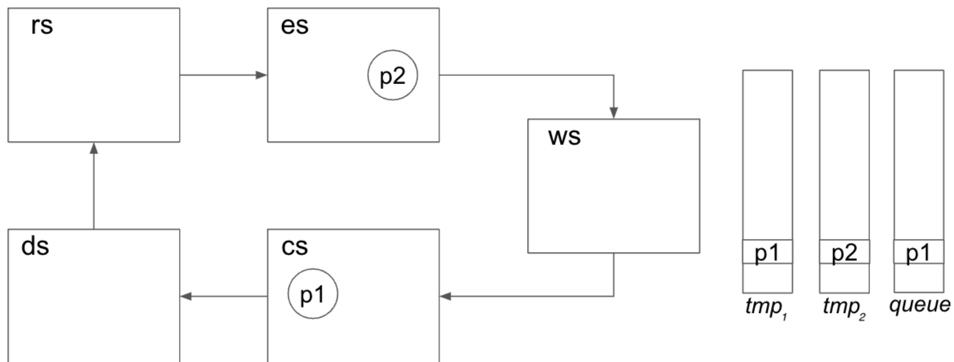


図 3.5: p1 が図 3.4 から状態遷移 3 を行った状態

p1 は cs へ移動した。図 3.5 から p1 の時と同様に p2 が状態遷移 2 を行った場合は下記になる。

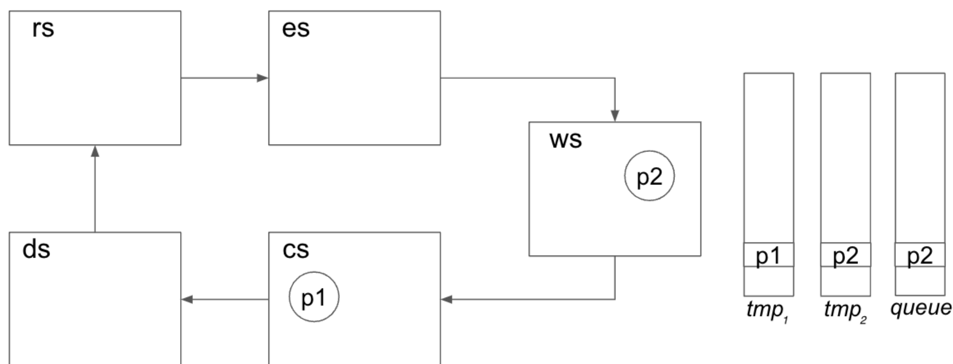


図 3.6: p2 が図 3.5 から状態遷移 2 を行った状態

p2 は ws へ移動したと同時に tmp_2 の待ち行列を共有変数 $queue$ へ代入した為 p2 と変わっている。続いて同様に p2 は ws にいて、且つ共有変数 $queue$ の待ち行列の先頭が p2 である為、p2 が状態遷移 3 を行える。図 3.6 から p2 が状態遷移 3 を行った場合は下記になる。

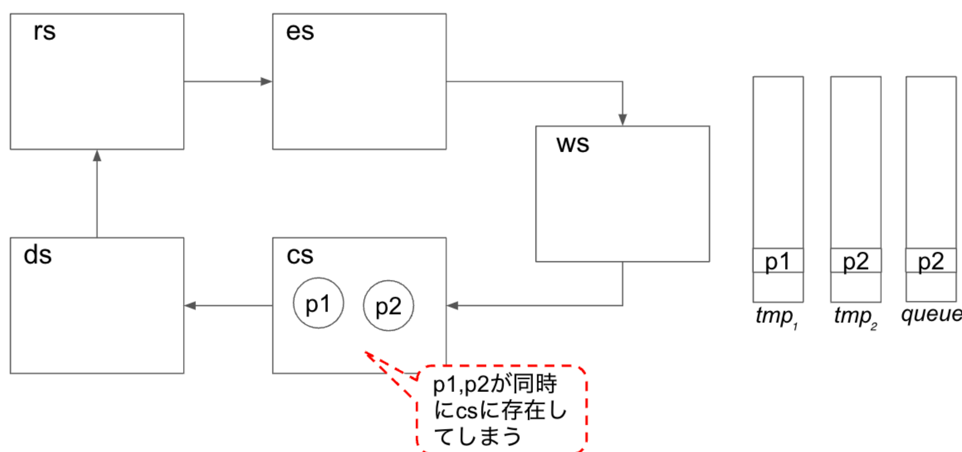


図 3.7: p2 が図 3.6 から状態遷移 3 を行った状態

ここの状態で p1 と p2 が同時に cs に存在している事が確認でき、相互排除性を満たしていない事がわかる。初期状態の図 3.1 から最後状態の図 3.7 に至る状態遷移は相互排除性を満たさない事の反例になっている。

3.1.3 観測可能成分

ここで使う観測可能成分の解説をする。

観測可能成分	説明
(pc[I]: L)	I はプロセス ID。L はラベルで rs,es,ws,cs,ds のいずれかが入る。 プロセス I は L の位置に存在しているという意味になる。
(tmp[I]: $Q1$)	I はプロセス ID。$Q1$ はプロセスの局所変数に格納されている 待ち行列を示す。 プロセス I の局所変数に待ち行列 Q1 が 格納されているという意味になる。
(queue: Q)	Q は共有変数に格納されている待ち行列を示す。 共有変数 *queue* に待ち行列 Q が格納されているという意味になる。

3.1.4 状態機械としての形式化

ここでは FQlock の状態機械を定義する。状態の集合 S は下記のように定義する。

$S \triangleq \{$
  $\{(\mathrm{pc[p1]}:\ l_1)\ (\mathrm{pc[p2]}:\ l_2)\ (\mathrm{queue}:\ q)\ (\mathrm{tmp[p1]}:\ q_1)\ (\mathrm{tmp[p2]}:\ q_2)\}$
  $\mid\ l_1, l_2 \in \mathrm{Loc},\ q, q_1, q_2 \in \mathrm{Queue}$
$\}$

初期状態の集合 I については下記のように定義する。

$I \triangleq \{\{(\mathrm{pc[p1]}:\ \mathrm{rs})\ (\mathrm{pc[p2]}:\ \mathrm{rs})\ (\mathrm{queue}:\ \mathrm{empty})\ (\mathrm{tmp[p1]}:\ \mathrm{empty})\ (\mathrm{tmp[p2]}:\ \mathrm{empty})\}\}$

状態遷移の集合 T については下記のように定義する。

$T \triangleq \{($
  $\{(\mathrm{pc}[P]:\ \mathrm{rs})\ (\mathrm{tmp}[P]:\ \mathit{Q1})\ (\mathrm{queue}:\ Q)\ \mathit{OCs}\},$
  $\{(\mathrm{pc}[P]:\ \mathrm{es})\ (\mathrm{tmp}[P]:\ \mathrm{enq}(Q, P))\ (\mathrm{queue}:\ Q)\ \mathit{OCs}\}$
$)\ \mid\ P \in \mathrm{Pid}\ ,\ Q, \mathit{Q1} \in \mathrm{Queue}\ ,\ \mathit{OCs} \in \mathrm{OComp}$
$\} \cup \{($
  $\{(\mathrm{pc}[P]:\ \mathrm{es})\ (\mathrm{tmp}[P]:\ \mathit{Q1})\ (\mathrm{queue}:\ Q)\ \mathit{OCs}\},$
  $\{(\mathrm{pc}[P]:\ \mathrm{ws})\ (\mathrm{tmp}[P]:\ \mathit{Q1})\ (\mathrm{queue}:\ \mathit{Q1})\ \mathit{OCs}\}$
$)\ \mid\ P \in \mathrm{Pid}\ ,\ Q, \mathit{Q1} \in \mathrm{Queue}\ ,\ \mathit{OCs} \in \mathrm{OComp}$
$\} \cup \{($
  $\{(\mathrm{pc}[P]:\ \mathrm{ws})\ (\mathrm{queue}:\ (P\ \ Q))\ \mathit{OCs}\},$
  $\{(\mathrm{pc}[P]:\ \mathrm{cs})\ (\mathrm{queue}:\ (P\ \ Q))\ \mathit{OCs}\}$
$)\ \mid\ P \in \mathrm{Pid}\ ,\ Q \in \mathrm{Queue}\ ,\ \mathit{OCs} \in \mathrm{OComp}$
$\} \cup \{($
  $\{(\mathrm{pc}[P]:\ \mathrm{cs})\ (\mathrm{tmp}[P]:\ \mathit{Q1})\ (\mathrm{queue}:\ Q)\ \mathit{OCs}\},$
  $\{(\mathrm{pc}[P]:\ \mathrm{ds})\ (\mathrm{tmp}[P]:\ \mathrm{deq}(Q))\ (\mathrm{queue}:\ Q)\ \mathit{OCs}\}$
$)\ \mid\ P \in \mathrm{Pid}\ ,\ Q, \mathit{Q1} \in \mathrm{Queue}\ ,\ \mathit{OCs} \in \mathrm{OComp}$
$\} \cup \{($
  $\{(\mathrm{pc}[P]:\ \mathrm{ds})\ (\mathrm{tmp}[P]:\ \mathit{Q1})\ (\mathrm{queue}:\ Q)\ \mathit{OCs}\},$
  $\{(\mathrm{pc}[P]:\ \mathrm{rs})\ (\mathrm{tmp}[P]:\ \mathit{Q1})\ (\mathrm{queue}:\ \mathit{Q1})\ \mathit{OCs}\}$
$)\ \mid\ P \in \mathrm{Pid}\ ,\ Q, \mathit{Q1} \in \mathrm{Queue}\ ,\ \mathit{OCs} \in \mathrm{OComp}$
$\}$

3.1.5 Maudeによる形式仕様

ここでは形式化されたものをどのようにMaudeで記述するかについて説明し、Maudeのsearchコマンドによるモデル検査を行う。プロセスが移動する5つの領域とプロセスを下記のLABELモジュールとPIDモジュールに記述した。

```
1 fmod LABEL is
2   sort Label .
3   ops rs es ws cs ds : -> Label .
4 endfm
5
6 fmod PID is
7   sort Pid .
8   ops p1 p2 : -> Pid .
9 endfm
```

次にパラメータモジュールを活用し、最初のビューはTRIVモジュールのソートEltをPIDモジュールのソートPidに置き換えてを行う。2番目のビューはTRIVモジュールのソートEltをOCOMPモジュールのソートOCompに置き換えを行う。

```
1 view Pid from TRIV to PID is
2   sort Elt to Pid .
3 endv
4
5 view OComp from TRIV to OCOMP is
6   sort Elt to OComp .
7 endv
```

次に結合法則、交換法則を満たす状態の集合を扱えるように汎用的なSOUPモジュールを記述する。

```
1 fmod SOUP{D :: TRIV} is
2   sort Soup{D} .
3   subsort D$Elt < Soup{D} .
4   op empty : -> Soup{D} [ctor] .
5   op _ _ : Soup{D} Soup{D} -> Soup{D} [ctor assoc comm id: empty] .
6 endfm
```

QUEUEモジュールではPidの識別子を要素とする待ち行列に対して末尾に識別子を追加する機能(enq)と先頭の識別子を取得する機能(top)と先頭の識別子を削除する機能(deq)を備えたQUEUEモジュールを記述する。

```
1  fmod QUEUE{D :: TRIV} is
2    sort Queue{D} .
3    subsort D$Elt < Queue{D} .
4    op empty : -> Queue{D} [ctor] .
5    op _ _ : D$Elt Queue{D} -> Queue{D} [ctor] .
6    op enq : Queue{D} D$Elt -> Queue{D} .
7    op deq : Queue{D} -> Queue{D} .
8    op top : Queue{D} -> D$Elt .
9    var Q : Queue{D} .
10   vars X Y : D$Elt .
```

```
11    eq enq(empty,X) = X empty .
12    eq enq((Y Q),X) = Y enq(Q,X) .
13    eq deq(empty) = empty .
14    eq deq((X Q)) = Q .
15    eq top((X Q)) = X .
16  endfm
```

OCOMP モジュールでは LABEL モジュール、Pid をパラメータとして与えた QUEUE モジュールをインポートして活用し、pc,queue,tmp の 3 つの観測可能成分を記述した。

```
1  fmod OCOMP is
2    pr LABEL .
3    pr QUEUE{Pid} .
4    sort OComp .
5    op (pc[_]:_) : Pid Label -> OComp [ctor] .
6    op (queue:_) : Queue{Pid} -> OComp [ctor] .
7    op (tmp[_]:_) : Pid Queue{Pid} -> OComp [ctor] .
8  endfm
```

CONFIG モジュールでは初期状態を記述した。初期状態は p1 と p2 プロセスが共に rs に位置して共有変数 *queue* は空、p1 の局所変数 tmp_1 と p2 の局所変数 tmp_2 は空である。また init で参照できるようにした。

```
1  fmod CONFIG is
2    pr SOUP{OComp} .
3    sort Config .
4    op {_} : Soup{OComp} -> Config [ctor] .
5    op init : -> Config .
6    eq init = {(pc[p1]: rs) (pc[p2]: rs) (queue: empty) (tmp[p1]: empty) (tmp[p2
        ]: empty)} .
7  endfm
```

QLOCK モジュールでは状態から状態への遷移を記述したモジュールである。3.1.1 の所で記載した状態遷移 1〜5 の 5 つ存在する。書き換え規則の eq1 は状態遷移 1、eq2 は状態遷移 2、wt は状態遷移 3、dq1 は状態遷移 4、dq2 は状態遷移 5 の動作を表している。

```
1  mod QLOCK is
2    pr CONFIG .
3    var OCs : Soup{OComp} .
4    vars Q R : Queue{Pid} .
5    var I : Pid .
6    vars L1 L2 : Label .
7    rl [eq1] : {(pc[I]: rs) (queue: Q) (tmp[I]: R) OCs}
8       => {(pc[I]: es) (queue: Q) (tmp[I]: enq(Q,I)) OCs} .
9    rl [eq2] : {(pc[I]: es) (queue: Q) (tmp[I]: R) OCs}
10      => {(pc[I]: ws) (queue: R) (tmp[I]: R) OCs} .
11   rl [wt] : {(pc[I]: ws) (queue: (I Q)) OCs}
12      => {(pc[I]: cs) (queue: (I Q)) OCs} .
13   rl [dq1] : {(pc[I]: cs) (queue: Q) (tmp[I]: R) OCs}
14      => {(pc[I]: ds) (queue: Q) (tmp[I]: deq(Q)) OCs} .
```

```
15    rl [dq2] : {(pc[I]: ds) (queue: Q) (tmp[I]: R) OCs}
16       => {(pc[I]: rs) (queue: R) (tmp[I]: R) OCs} .
17  endm
```

3.1.6 Maude の search コマンドによる不変性モデル検査

FQlock が相互排除性を満たすかどうかのモデル検査のために用いた Search コマンドは下記である。

```
search [1] in QLOCK : init =>* {(pc[p1]: cs) (pc[p2]: cs) OCs} .
```

init は CONFIG モジュールで定義した初期状態を表す項である。(pc[p1]: cs) (pc[p2]: cs) はプロセス p1 と p2 が同時に cs にいる状態を意味しており、その状態を探索する。

実行した結果は下記になる。

```
Maude> search [1] in QLOCK : init =>* {(pc[p1]: cs) (pc[p2]: cs) OCs} .
search [1] in QLOCK : init =>* {OCs (pc[p1]: cs) pc[p2]: cs} .

Solution 1 (state 33)
states: 34  rewrites: 66 in 0ms cpu (0ms real) (183844 rewrites/second)
OCs --> queue: (p2 empty) (tmp[p1]: p1 empty) tmp[p2]: p2 empty
```

ソリューションが 1 つ出力され、state 33 でプロセス p1,p2 が同時に cs にいる状態が発見された。state 33 に至るまでの経路を詳しく見てみるには show path コマンドで可能にであり、下記は show path コマンドの結果になる。

```
Maude> show path 33 .
state 0, Config: {queue: empty (pc[p1]: rs) (pc[p2]: rs) (tmp[p1]: empty) tmp[p2]:
    empty}
===[ rl {OCs queue: Q (pc[I]: rs) tmp[I]: R} => {(pc[I]: es) queue: Q OCs tmp[I]:
    enq(Q, I)} [label eq1] . ]===>
state 1, Config: {queue: empty (pc[p1]: es) (pc[p2]: rs) (tmp[p1]: p1 empty) tmp[p2
    ]: empty}
===[ rl {OCs queue: Q (pc[I]: rs) tmp[I]: R} => {(pc[I]: es) queue: Q OCs tmp[I]:
    enq(Q, I)} [label eq1] . ]===>
state 3, Config: {queue: empty (pc[p1]: es) (pc[p2]: es) (tmp[p1]: p1 empty) tmp[p2
    ]: p2 empty}
===[ rl {OCs queue: Q (pc[I]: es) tmp[I]: R} => {(pc[I]: ws) queue: R OCs tmp[I]: R
    } [label eq2] . ]===>
state 6, Config: {queue: (p1 empty) (pc[p1]: ws) (pc[p2]: es) (tmp[p1]: p1 empty)
    tmp[p2]: p2 empty}
===[ rl {OCs queue: (I Q) pc[I]: ws} => {(pc[I]: cs) OCs queue: (I Q)} [label wt] .
    ]===>
state 13, Config: {queue: (p1 empty) (pc[p1]: cs) (pc[p2]: es) (tmp[p1]: p1 empty)
    tmp[p2]: p2 empty}
===[ rl {OCs queue: Q (pc[I]: es) tmp[I]: R} => {(pc[I]: ws) queue: R OCs tmp[I]: R
    } [label eq2] . ]===>
state 23, Config: {queue: (p2 empty) (pc[p1]: cs) (pc[p2]: ws) (tmp[p1]: p1 empty)
    tmp[p2]: p2 empty}
```

```
===[ rl {OCs queue: (I Q) pc[I]: ws} => {(pc[I]: cs) OCs queue: (I Q)} [label wt] .
    ]===>
state 33, Config: {queue: (p2 empty) (pc[p1]: cs) (pc[p2]: cs) (tmp[p1]: p1 empty)
    tmp[p2]: p2 empty}
```

わかりやすく図示したものが下記になる。

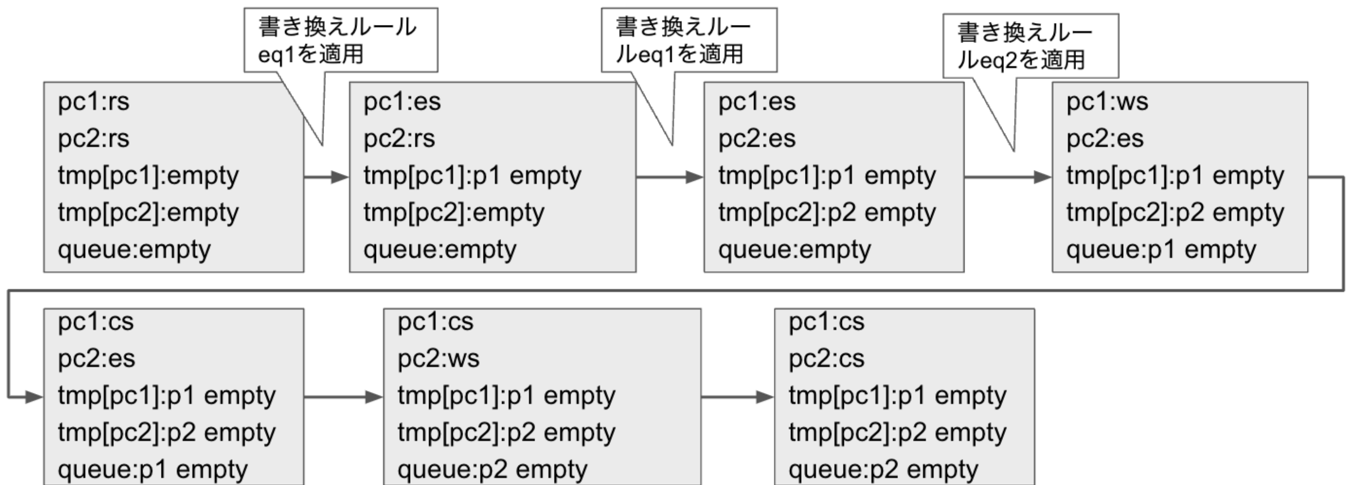


図 3.8: state 33 までの経路

図 3.8 の一番左は初期状態になる。順に書き換えルールを適用しプロセス 1,2 が同時に cs に存在することがわかる。

3.1.7 Queue の値が積み上がる構造

ここでは相互排除性を満たさない Qlock のモデル検査を通じて発見したことを記す。共有変数 *queue* の値が無限に積み上がる構造になっており、到達可能状態の全探索が不可能であることに気づいたので下記の図を元に説明をする。

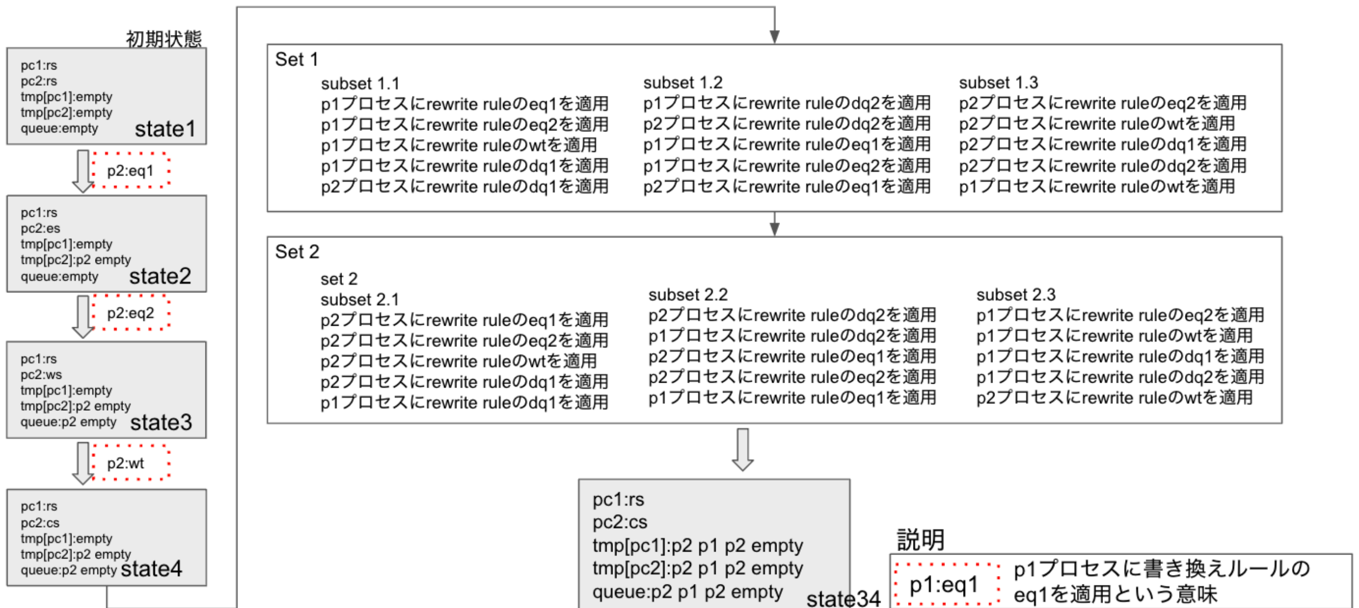


図 3.9: Queue 変数が積み上がる遷移

左上の state1 は初期状態である。初期状態から p2 プロセスに eq1,eq2,wt の遷移をして state4 の状態に遷移する。state4 の状態から下記の書き換え規則を順に実行し state34 の状態へ遷移する。

1. set1

   (a) p1 プロセスに rewrite rule の eq1 を適用

   (b) p1 プロセスに rewrite rule の eq2 を適用

   (c) p1 プロセスに rewrite rule の wt を適用

   (d) p1 プロセスに rewrite rule の dq1 を適用

   (e) p2 プロセスに rewrite rule の dq1 を適用

   (f) p1 プロセスに rewrite rule の dq2 を適用

   (g) p2 プロセスに rewrite rule の dq2 を適用

   (h) p1 プロセスに rewrite rule の eq1 を適用

   (i) p1 プロセスに rewrite rule の eq2 を適用

   (j) p2 プロセスに rewrite rule の eq1 を適用

   (k) p2 プロセスに rewrite rule の eq2 を適用

   (l) p2 プロセスに rewrite rule の wt を適用

   (m) p2 プロセスに rewrite rule の dq1 を適用

(n) p2 プロセスに rewrite rule の dq2 を適用

(o) p1 プロセスに rewrite rule の wt を適用

2. set 2

(a) p2 プロセスに rewrite rule の eq1 を適用

(b) p2 プロセスに rewrite rule の eq2 を適用

(c) p2 プロセスに rewrite rule の wt を適用

(d) p2 プロセスに rewrite rule の dq1 を適用

(e) p1 プロセスに rewrite rule の dq1 を適用

(f) p2 プロセスに rewrite rule の dq2 を適用

(g) p1 プロセスに rewrite rule の dq2 を適用

(h) p2 プロセスに rewrite rule の eq1 を適用

(i) p2 プロセスに rewrite rule の eq2 を適用

(j) p1 プロセスに rewrite rule の eq1 を適用

(k) p1 プロセスに rewrite rule の eq2 を適用

(l) p1 プロセスに rewrite rule の wt を適用

(m) p1 プロセスに rewrite rule の dq1 を適用

(n) p1 プロセスに rewrite rule の dq2 を適用

(o) p2 プロセスに rewrite rule の wt を適用

state34 の状態から set1,set2 の遷移を適用すると下記の図のように共有変数 *queue* の値が積み上がって行く事が確認できる。

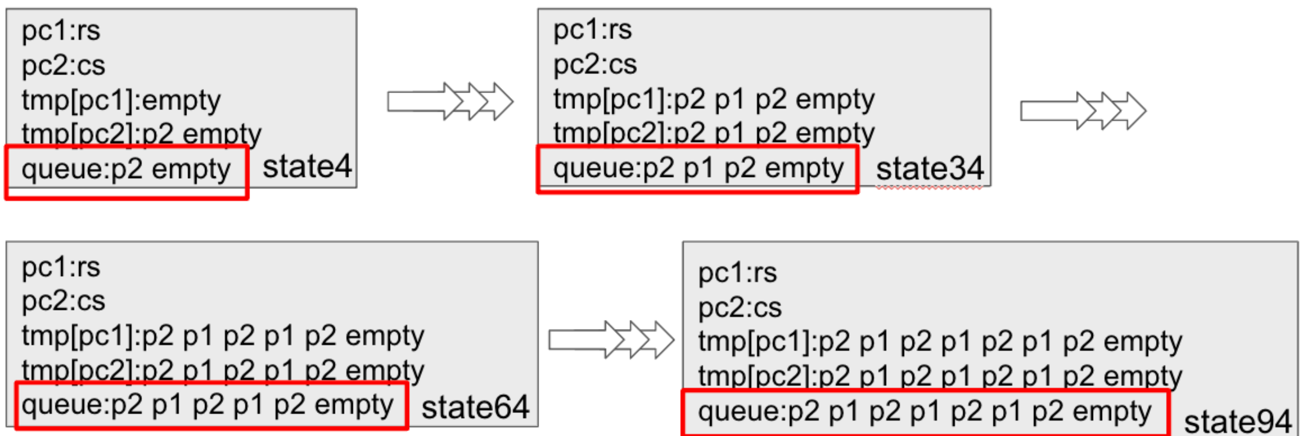


図 3.10: 共有変数 *queue* が積み上がる

このように $queue$ の値が積み上がり続け到達可能状態が無限に作られる状況においても Maude のモデル検査で反例を見つける事ができ、正常に終了ができた。Maude のモデル検査が幅優先で探索している為である事がわかった。

3.2 FQlockの修正

FQlockでは待ち行列の末尾にプロセスの識別子を追加する時に不可分に行っていない為、相互排除性を満たさない。不可分に行っていない事により各プロセスが参照する共有変数 *queue* の値がそれぞれで異なる現象が起きる。その為複数プロセスがcsに存在する時がある。FQlockの修正として下記2点を行う。

- 待ち行列の末尾へプロセス識別子の追加を不可分に行う
- 待ち行列の先頭のプロセス識別子の削除を不可分に行う

ここからはFQlockを修正し、相互排除性を満たすものをQlockと呼ぶ。

3.2.1 疑似コード

ここでQlockの疑似コードを書きに示す。

4 疑似コード Qlock

```
1: Loop {
2: "Remainder Section"
3: rs: atomic_enqueue(queue,i);
4: ws: repeat until top(queue)=i;
5: "Critical Section"
6: cs: atomic_dequeue(queue);
7: }
```

疑似コードから下記の状態遷移が発生することが分かる。プロセスp1と限定して記載しているが、実際にはプロセスp1に限らず、FQlockに参加しているいかなるプロセスも対象となる

1. 状態遷移1
   p1がrsにいればwsに移動すると共にプロセス識別子の待ち行列である *queue* の末尾にp1の識別子を追加する
2. 状態遷移2
   p1がwsでプロセス識別子の待ち行列である *queue* の先頭がp1であればcsに移動する
3. 状態遷移3
   p1がcsでrsに移動すると共にプロセス識別子の待ち行列である *queue* の先頭の値を削除する

3.2.2 図解

ここでは疑似コードから p1 プロセス、p2 プロセスの 2 つが存在していた場合の遷移の一例を図で説明する。矢印はプロセスの移動方向を表す。p1 と p2 はプロセスを表す。tmp_1 は p1 プロセスの局所変数を表す。tmp_2 は p2 プロセスの局所変数を表す。$queue$ は共有変数を表す。

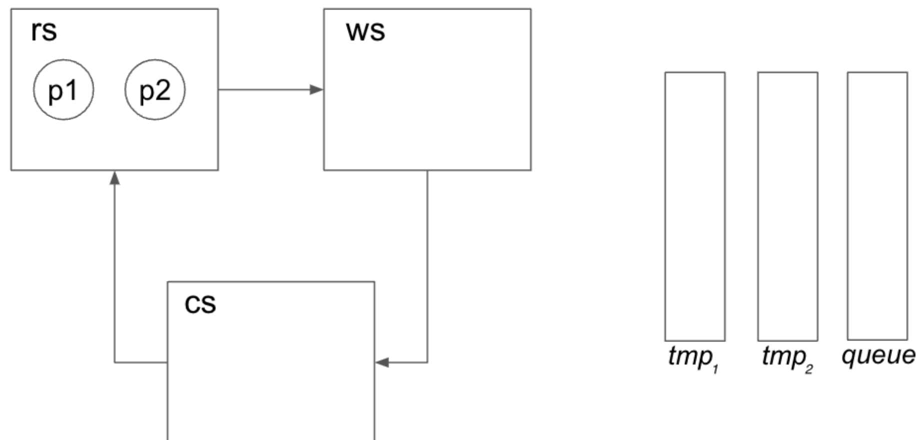


図 3.11: 初期状態

p1 と p2 プロセスは初期状態なので共に rs にいる。それぞれの局所変数と共有変数は空である。図 3.11 の初期状態からプロセス p1 が状態遷移 1 を行った場合は下記になる。

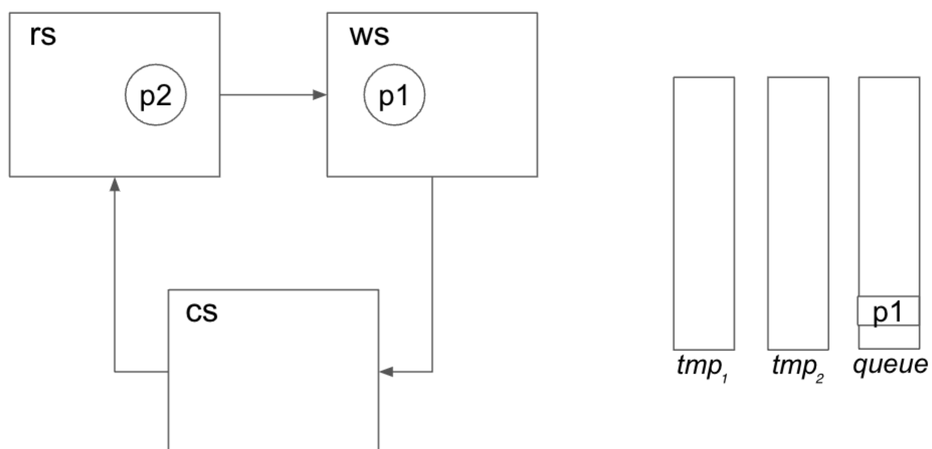


図 3.12: p1 が図 3.11 から状態遷移 1 を行った状態

p1 は ws へ移動したと同時に空の待ち行列 $queue$ の末尾に p1 の識別子を追加された。続いて図 3.12 からプロセス p2 が状態遷移 1 を行った場合は下記になる。

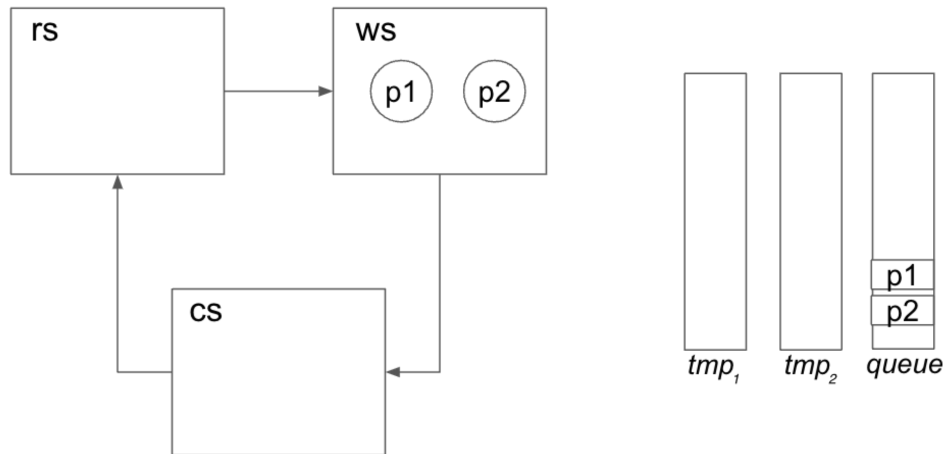


図 3.13: p2 が図 3.12 から状態遷移 1 を行った状態

p1 と同様に p2 が ws に移動した同時に待ち行列である共有変数 $queue$ の末尾に p2 プロセスの識別子が追加された。次に図 3.13 からプロセス p1 が状態遷移 2 を行った場合は下記になる。

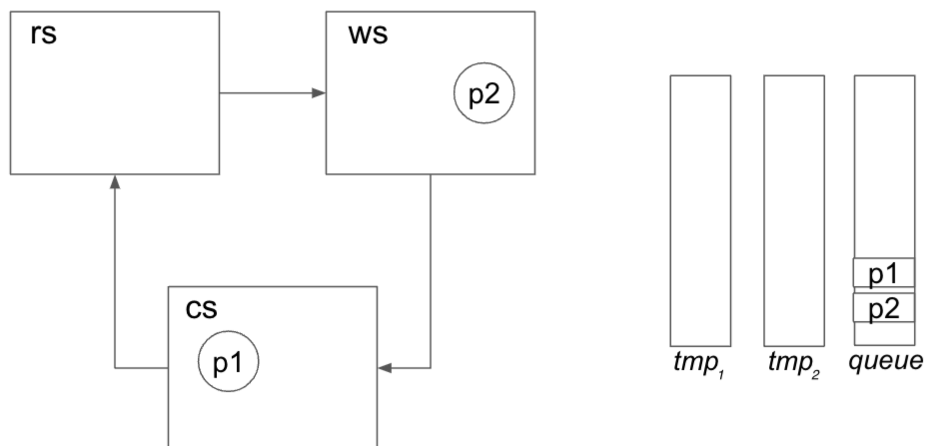


図 3.14: p1 が図 3.13 から状態遷移 2 を行った状態

待ち行列である共有変数 $queue$ の先頭が p1 プロセスの識別子である為、p1 プロセスは cs へ移動した。図 3.14 からプロセス p1 が状態遷移 3 を行った場合は下記になる。

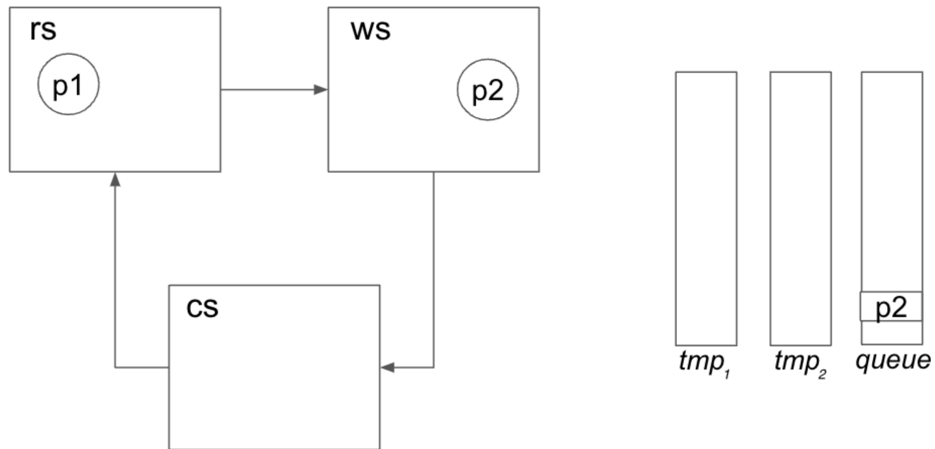


図 3.15: p1 が図 3.14 から状態遷移 3 を行った状態

p1 が rs へ移動したと同時に待ち行列である共有変数 $queue$ の先頭 (p1 プロセスの識別子) を削除した。次に図 3.15 からプロセス p2 が状態遷移 2 を行った場合は下記になる。

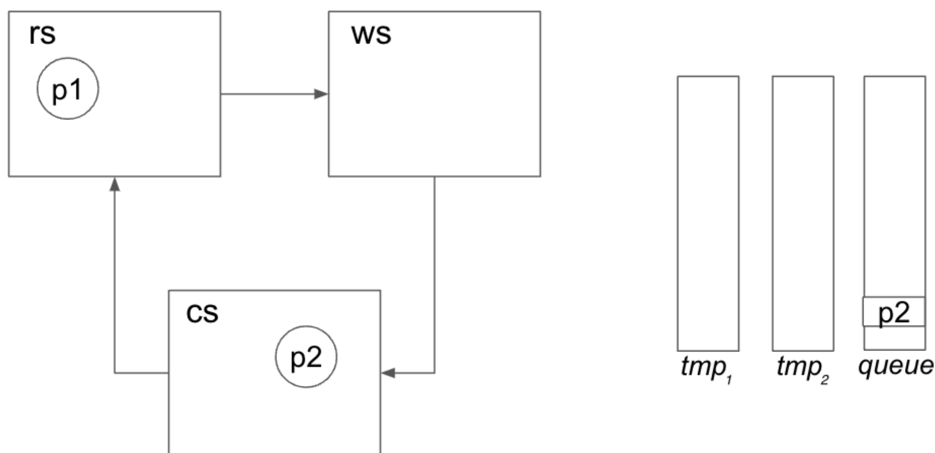


図 3.16: p2 が図 3.15 から状態遷移 2 を行った状態

以前と同様に待ち行列である共有変数 $queue$ の先頭が p2 プロセスの識別子である為、p2 プロセスは cs へ移動した。図 3.16 からプロセス p2 が状態遷移 3 を行った場合は下記になる。

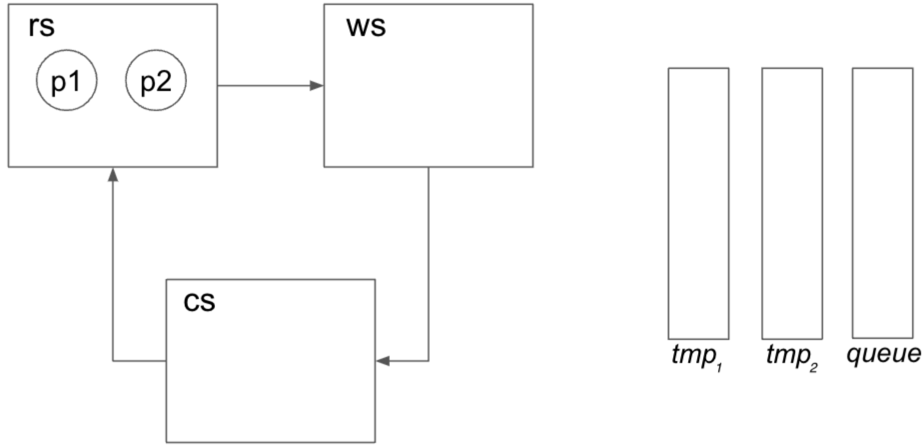


図 3.17: p2 が図 3.16 から状態遷移 3 を行った状態

p2 が rs へ移動したと同時に待ち行列である共有変数 $queue$ の先頭 (p2 プロセスの識別子) を削除した。ここまで p1 の局所変数 tmp_i と p2 の局所変数 tmp_2 は一度も使用されない。3 つ以上の複数プロセスになっても同様である。待ち行列である共有変数 $queue$ に末尾への追加、先頭の削除を不可分に行っており、局所変数を利用する必要がない為である。

3.2.3 状態機械としての形式化

ここでは Qlock の状態機械を定義する。状態の集合 S は下記のように定義する。

S ≜ {
{(pc[p1]: l_1) (pc[p2]: l_2) (queue: q)}
| p1,p2 ∈ Pid, l_1,l_2 ∈ Loc, q ∈ Queue
}

初期状態の集合 I については下記のように定義する。

I ≜ {{(pc[p1]: rs) (pc[p2]: rs) (queue: empty)}}

状態遷移の集合 T については下記のように定義する。

T ≜ {(
{(pc[P]: rs) (queue: Q) OCs},
{(pc[P]: ws) (queue: enq(Q,P)) OCs}
) | P ∈ Pid , Q ∈ Queue , OCs ∈ OComp
} ∪ {(
{(pc[P]: ws) (queue: (P Q)) OCs},
{(pc[P]: cs) (queue: (P Q)) OCs}
) | P ∈ Pid , Q ∈ Queue , OCs ∈ OComp
} ∪ {(
{(pc[P]: cs) (queue: Q) OCs},

```
  {(pc[P]: rs) (queue: deq(Q)) OCs}
) | P ∈ Pid , Q ∈ Queue , OCs ∈ OComp
}
```

3.2.4 Maudeによる形式仕様

ここでは形式化されたQlockをどのようにMaudeで記述するかについて説明し、Maudeのsearchコマンドによるモデル検査を行う。FQlockの3.1.5の記述とほぼ同じになるが書き換えルールが異なる為、その部分を下記に示す。

```
1  mod QLOCK is
2    pr CONFIG .
3    var OCs : Soup{OComp} .
4    vars Q R : Queue{Pid} .
5    var I : Pid .
6    vars L1 L2 : Label .
7    rl [eq] : {(pc[I]: rs) (queue: Q) OCs} => {(pc[I]: ws) (queue: enq(Q,I)) OCs}
       .
8    rl [wt] : {(pc[I]: ws) (queue: (I Q)) OCs} => {(pc[I]: cs) (queue: (I Q)) OCs
       } .
9    rl [dq] : {(pc[I]: cs) (queue: Q) OCs} => {(pc[I]: rs) (queue: deq(Q)) OCs} .
10 endm
```

3.2.1の所で記載した状態遷移1〜3の3つ存在する。書き換え規則のeqは状態遷移1の動作を表し、待ち行列である*queue*の末尾にプロセスの識別子を不可分に追加している。wtは状態遷移2の動作を表し、待ち行列*queue*の先頭がプロセスの識別子であればcsへ移動している。dqは状態遷移3の動作を表し、待ち行列*queue*の先頭の値の削除を不可分に行っている。

3.2.5 Maudeのsearchコマンドによる不変性モデル検査

Qlockが相互排除性を満たすかどうかのモデル検査のために用いたSearch コマンドは下記である。

```
search [1] in QLOCK : init =>* {(pc[p1]: cs) (pc[p2]: cs) OCs} .
```

実行した結果は下記になる

```
Maude> search [1] in QLOCK : init =>* {(pc[p1]: cs) (pc[p2]: cs) OCs} .
search [1] in QLOCK : init =>* {OCs (pc[p1]: cs) pc[p2]: cs} .

No solution.
states: 9  rewrites: 29 in 0ms cpu (0ms real) (164772 rewrites/second)
```

相互排除性が満たされた為にp1とp2プロセスが同時にcsにいる状態が到達可能状態において1つも存在しなかった。その為にNo solutionという結果になった。また今回は全ての状態が9つとなり、図解可能である為下記に示す。

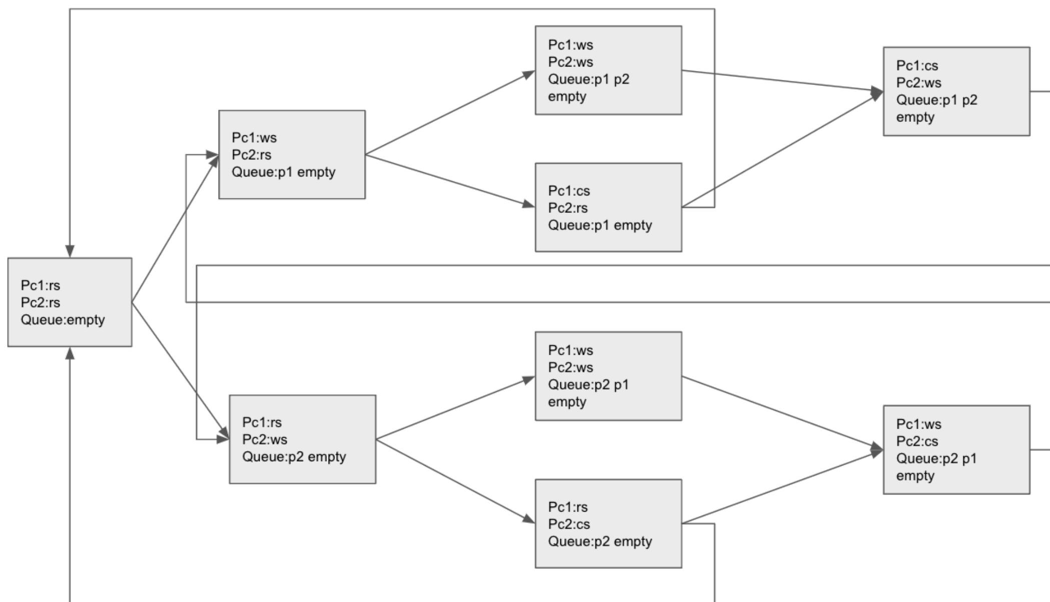


図 3.18: 初期状態から始まる全ての到達可能状態

矢印は状態から状態への遷移を示しており、p1 と p2 が同時に cs にいる状態がない事がわかる。

3.3 プロセス数を増やした場合の Qlock

ここではプロセス数を 3 つ以上に増やした場合の search コマンドによるモデル検査の実行時間がどのように変化するかに着目する。Maude の記述の変更を下記に記す。(プロセスが 8 つの場合を実験する際の例になる)

プロセスを扱う PID モジュールに扱うプロセス分記載する。

```
fmod PID is
  sort Pid .
  ops p1 p2 p3 p4 p5 p6 p7 p8 : -> Pid .
endfm
```

初期状態を表す CONFIG モジュールの init 機能でそれぞれのプロセスの最初の位置を記述する。全ての初期状態は rs にいる。

```
fmod CONFIG is
  pr SOUP{OComp} .
  sort Config .
  op {_} : Soup{OComp} -> Config [ctor] .
  op init : -> Config .
  eq init = {(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)} .
endfm
```

search コマンドを実行する際に初期状態からすべてのプロセスが cs にいる状態を探索する。

```
search [1] in QLOCK : init =>* {(pc[p1]: cs) (pc[p2]: cs) (pc[p3]: cs) (pc[p4]:
    cs) (pc[p5]: cs) (pc[p6]: cs) (pc[p7]: cs) (pc[p8]: cs) OCs} .
```

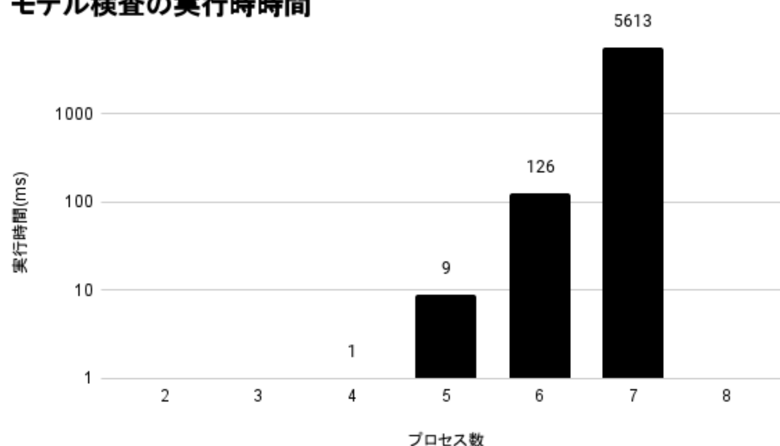
3.3.1 実験結果

実験に用いた環境は下記の通りである。

- MacBook Pro 16-inch, 2019
- macOS Ventura version13.5
- 2.3 GHz 8-Core Intel Core i9
- 32 GB 2667 MHz DDR4

プロセス数が 7 個までは現実的な時間内で実行が終了したが、8 個以上になると終了できなかった。下記はプロセス数を順に増やした時のモデル検査の実行時間をまとめた表とグラフである。

プロセス数	2	3	4	5	6	7	8
実行時間	0ms	0ms	1ms	9ms	126ms	5613ms	NA



プロセス数が 8 個の時は定義した観測可能成分から pc[p1]〜pc[p8] の 8 つの変数を扱う事になる。変数の数が増えれば変数が取りうる値（rs,ws,cs）の数を変数分

掛け合わせた数の状態数に増える為、膨大になる。このように指数的に状態数が増え、状態空間爆発が起こり、モデル検査が終了できなかったと考えられる。

3.3.2 既存研究

ここでは状態空間爆発の課題を緩和する既存研究について記述する。その一つに分割統治法に基づいた Divide & Conquer Approach(DCA) が存在する。DCA に関連した提案はいくつか存在する [2][3][4]。これらの既存研究では leads-to プロパティのモデル検査において状態空間爆発問題を緩和する為の提案になっている。Leads-to プロパティとはシステムあるいはプロトコルがある状態になったらそのうち必ず他の状態になるといった性質である。QLock を例にすると、あるプロセスが ws にいるならば、そのプロセスは最終的には必ず cs に入る事ができるという事を意味する。ただ単に分割するだけでは不十分であり、分割した問題に対して何を検査すれば分割前の問題の検査と等価になるか難しいポイントである。

第4章　Identity-Friend-or-Foe 認証プロトコル (IFF)

IFF はグループのメンバーであるかどうかを確認する認証プロトコルである。ここでは 2 通りの IFF を題材にしてどのように状態機械として形式化したか、どのように Maude で記述したか、モデル検査した保証すべき性質の内容、それに対する反例の説明を行う。IFF は下記の前提条件を元に行われる。

1. グループごとに固有の鍵があり、メンバーのみがその鍵を知っている。
2. グループごとの固有の鍵は他のグループメンバーに漏えい、暗号を解読される事はない。

IFF プロトコルは、次の 2 つのメッセージのやり取りとして記述することができる。

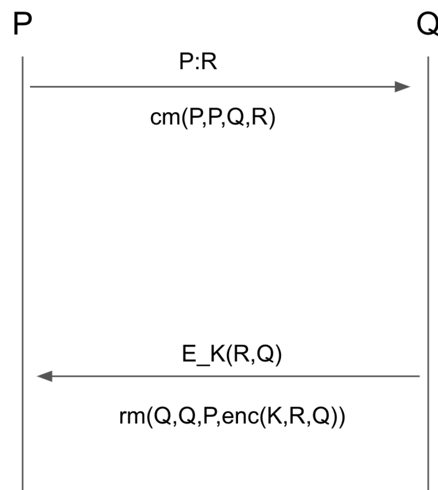


図 4.1: IFF メッセージ交換

P と Q が存在していて P は Q が同じ仲間であるか確かめたいとする。P は乱数の R を生成し、Q にメッセージ送信する。Q はメッセージを受信すると P の乱数と Q の ID を共通の鍵で暗号化してメッセージを返信する。P は Q からの返信を受信すると共通の鍵を利用して暗号文の復号を試みる。復号化に成功し、メッセー

ジにPが生成した乱数とQの識別子が存在していた場合、Qは同じメンバーであると認識する。これらのメッセージはネットワーク上に流れる。

4.1 観測可能成分

ここで使う観測可能成分の解説をする。

観測可能成分	説明
(nw: ms)	ms はネットワークに流れたメッセージのスープである 認証を行う principal のメッセージ交換するネットワークを表す
(prin: ps)	ps は principal のスープである 認証を行う登場人物になる
(rands: rs)	rs は乱数のスープである 各 principal が発行する乱数を表す

4.2 Maude におけるメッセージ送信/返信の表現

図 4.1 の cm(P,P,Q,R) がメッセージ送信の表現になり、rm(Q,Q,P,enc(K,R,Q)) がメッセージ返信の表現になり、それぞれの意味は下記の通りである。

cm(P,P,Q,R) ... cm(作成者, 送信者, 受信者, 乱数)
rm(Q,Q,P,enc(K,R,Q)) ... rm(作成者, 送信者, 受信者, 暗号文)

cm,rm 関数とも第一引数はメッセージの作者を表す。第二引数は送信者を表す。第三引数は受信者を表す。第四引数は乱数または暗号文を表す。第一引数の情報はメタ情報であり、第三者による改ざんは不可なものであるが、それ以外の引数は第三者によって改ざんが可能な情報である。暗号文の表現は下記の通りである。

enc(K,R,Q) ... enc(利用する共通鍵, 乱数, 暗号実行者の ID)

第一引数は所属しているグループの共有鍵を表す。第二引数は乱数であり、cm で受け取った乱数を利用する。第三引数は暗号を実行する者の ID を表す。

4.3 敵味方の識別性を満たさない IFF

ここでは意図的に誤りを埋め込んだ FIFF(Flawed IFF) を題材にして状態機械としての形式化、Maude での記述方法、モデル検査での保証すべき性質の内容、それに対する反例の説明を行う。この FIFF では下記の誤りをあえて注入する

- 暗号化の際に実行者の識別子を含めずに暗号化を行う

具体的には下記のようになる

誤りのある暗号化 ... enc(K,R)
誤りのない暗号化 ... enc(K,R,Q)

K は利用する共通鍵、R は乱数、Q は認証相手であるメンバー Q である。この誤りにより敵味方の識別性を満たすことができず、誤認が完成してしまう。下記の図は識別性を満たさずに誤認が完成した時の概要図になる。

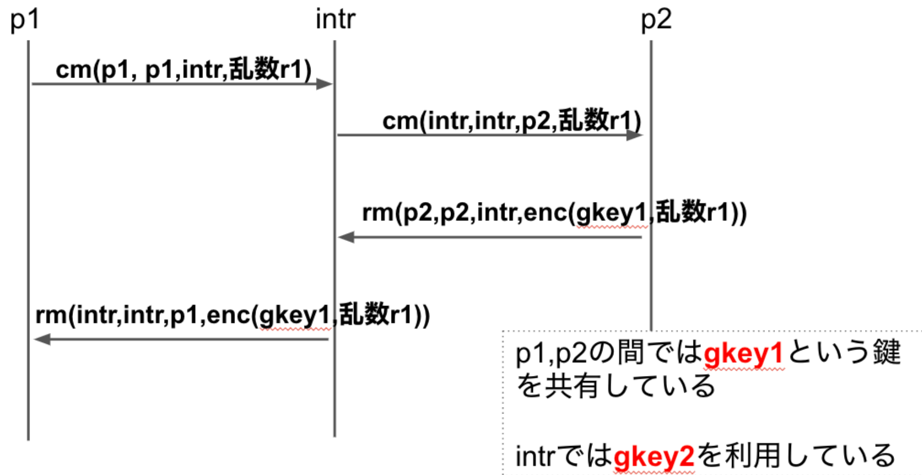


図 4.2: 誤認が成立する概要図

p1 は最初のメッセージを送る送信者、intr は侵入者（別グループのメンバー)、p2 はメッセージの返信者である。p1 と p2 は同じグループのメンバーであり、intr は別グループのメンバーである。p1 は認証を行う為に p1 の乱数を生成し、intr にメッセージを送る。intr は受け取ったメッセージに含まれる p1 の乱数を利用して p2 にメッセージを送る。p2 は共通鍵を用いて p1 の乱数を暗号化して intr へ返信する。intr は p2 からの返信メッセージを p1 へ送る。p1 は共有鍵を用いて復号化し、p1 が生成した乱数を確認し、intr を同じメンバーであると認識する。

4.3.1 状態機械としての形式化

ここではどのように状態機械へと落とし込むかについて説明する。状態の集合 S は下記のように定義する。

$$
\begin{aligned}
&S \triangleq \{ \\
&\quad \{(\mathrm{nw}:\ ms)\ (\mathrm{prins}:\ \mathrm{ps})\ (\mathrm{rands}:\ rs)\} \\
&\quad \mid\ ms \in \mathrm{Set(Msg)},\ \mathrm{ps} \in \mathrm{Set(Prin)},\ rs \in \mathrm{Set(Rand)} \\
&\}
\end{aligned}
$$

Msg、Prin、Rand はメッセージ、参加者、乱数のソートで、Set(Msg)、Set(Prin)、Set(Rand) の集合を意味する。

初期状態の集合 I は下記のように定義する。

```
I ≜ {{(nw: empty) (prins: (p1 p2 intr)) (rands: (r1 r2))}}
```

empty は空集合を意味する。参加者は p1,p2,intr の 3 つであり、p1 と p2 は同じメンバー、intr は別グループのメンバーを意味する。利用する乱数は 2 つである。

状態遷移の集合 T は下記のように定義する。

```
T ≜ {(
  {(nw: Ms) (prins: (P Q Ps)) (rands: (R Rs))},
  {(nw: (cm(P,P,Q,R) Ms)) (prins: (P Q Ps)) (rands: Rs)}
) | P,Q ∈ Prin, R ∈ Rand, Ms ∈ MsgSoup, Ps ∈ PrinSoup, Rs ∈ RandSoup
} ∪ {(
  {(nw: (cm(P2,P,Q,R) Ms)) OCs},
  {(nw: (rm(Q,P,enc(key(Q),R)) cm(P2,P,Q,R) Ms)) OCs}
) | P,P2,Q ∈ Prin, R ∈ Rand, Ms ∈ MsgSoup, OCs ∈ OComp
} ∪ {(
  {(nw: (cm(P3,P2,Q2,R) Ms)) (prins: (P Q Ps)) OCs},
  {(nw: (cm(intr,P,Q,R) cm(P3,P2,Q2,R) Ms)) (prins: (P Q Ps)) OCs}
) | P,P2,P3,Q,Q2,intr ∈ Prin, R ∈ Rand, Ms ∈ MsgSoup, Ps ∈ PrinSoup, OCs ∈
    OComp
} ∪ {(
  {(nw: (cm(P3,P2,Q2,R) Ms)) (prins: (P Q Ps)) OCs},
  {(nw: (rm(intr,Q,P,enc(key(intr),R)) cm(P3,P2,Q2,R) Ms)) (prins: (P Q Ps))
      OCs}
) | P,P2,P3,Q,Q2,intr ∈ Prin, R ∈ Rand, Ms ∈ MsgSoup, Rs ∈ RandSoup, OCs
    ∈ OComp
} ∪ {(
  {(nw: (rm(P3,P2,Q2,C) Ms)) (prins: (P Q Ps)) OCs},
  {(nw: (rm(intr,Q,P,C) rm(P2,P2,Q2,C) Ms)) (prins: (P Ps)) OCs}
) | P,P2,P3,Q,Q2 ∈ Prin, C ∈ Cipher, Ms ∈ MsgSoup, Ps ∈ PrinSoup, OCs ∈
    OComp
}
```

最初の遷移では P から Q へ最初のメッセージ送信を意味する。二番目の遷移では最初のメッセージの返信を Q から P へ送信を意味している。三番目の遷移では最初のメッセージ送信がネットワークに流れた後に intr が最初のメッセージの乱数 R をそのまま利用して別のメンバーへの送信を意味している。四番目の遷移では最初のメッセージ送信がネットワークに流れた後に intr が最初のメッセージの乱数 R と intr が持つ共通鍵を利用して返信メッセージを作成し、別のメンバーへの送信を意味している。最後の遷移では返信メッセージがネントワークに流れた後に返信メッセージの暗号文をそのまま利用して別のメンバーへの送信を意味している。

4.3.2 Maude による形式仕様

形式化されたものをどのようにMaudeで記述するかについて説明し、Maudeの search コマンドによるモデル検査を行う。Maudeのコードで記述したものが下記になる。ここでは主要な部分を抜粋して説明する。

p1とp2は同じ共有鍵を所有し、intrは別の共有鍵を所有していることを下記のKEYモジュールで記載した。

```
fmod KEY is
  pr PRIN .
  sort Key .
  ops gKey1 gKey2 : -> Key [ctor] .
  op key : Prin -> Key .
  eq key(p1) = gKey1 .
  eq key(p2) = gKey1 .
  eq key(intr) = gKey2 .
endfm
```

暗号化は共通鍵と乱数を利用して作成することを下記のCIPHERモジュールで記載した。

```
fmod CIPHER is
  pr PRIN .
  pr RAND .
  pr KEY .
  sort Cipher .
  op enc : Key Rand -> Cipher .
endfm
```

FLAWED-IFFモジュールでは遷移である書き換え規則を記載した。

```
mod FLAWED-IFF is
  pr OCOMPSET .
  sort Config .
  *** Config
  op {_} : OComp -> Config [ctor] .

  vars M : Msg .
  var NW : Soup{Msg} .
  var R : Rand .
  vars Rs R2s : Soup{Rand} .
  vars P P2 P3 Q Q2 Q3 : Prin .
  var Ps : Soup{Prin} .
  vars C : Cipher .
  var Cs : Soup{Cipher} .
  var OCs : Soup{OComp} .

rl [sdcm] : {(nw: NW) (prins: (P Q Ps)) (rands: (R Rs))}
    => {(nw: (cm(P, P,Q,R) NW)) (prins: (P Q Ps)) (rands: Rs)} .

rl [sdrm] : {(nw: (cm(P2, P,Q,R) NW)) OCs}
    => {(nw: (rm(Q, Q, P,enc(key(Q),R)) cm(P2, P,Q,R) NW)) OCs} .
```

```
rl [fakecm1] : {(nw: (cm(P3,P2,Q2,R) NW)) (prins: (P Q Ps)) OCs}
    => {(nw: (cm(intr,P,Q,R) cm(P3, P2,Q2,R) NW)) (prins: (P Q Ps)) OCs} .

rl [fakerm1] : {(nw: (cm(P3, P2,Q2,R) NW)) (prins: (P Q Ps)) OCs}
    => {(nw: (rm(intr, Q, P,enc(key(intr),R)) cm(P3,P2,Q2,R) NW)) (prins: (P Q
        Ps)) OCs} .

rl [fakerm2] : {(nw: (rm(P3, P2, Q2, C) NW)) (prins: (P Q Ps)) OCs}
    => {(nw: (rm(intr, Q, P,C) rm(P2, P2, Q2, C) NW)) (prins: (P Q Ps)) OCs} .
endm
```

- 書換え規則 sdcm は P から Q へ最初のメッセージ乱数を送信する。
- 書換え規則 sdrm は Q は受けとったメッセージの乱数に自身が持っている鍵を使って暗号化して返信をする。
- 書換え規則 Fakecm1 は侵入者はネットワークに流れる乱数を取得し、送り主、送り先を偽装して送信する。ただしメッセージ発行者は偽装できない。
- 書換え規則 Fakerm1 は侵入者はネットワークに流れる乱数を取得し、暗号文を作り、送り主、送り先を偽装してリプライメッセージを送る。ただしメッセージ発行者は偽装できない。
- 書換え規則 Fakerm2 は侵入者はネットワークに流れる暗号文を取得し、送り主、送り先を偽装してリプライメッセージを送る。ただしメッセージ発行者は偽装できない。

FLAWED-IFF-INIT モジュールでは初期状態を記載した。初期状態では nw は空であり、prins は p1 と p2 と intr が存在し、rands では r1 と r2 の 2 つの乱数が存在している。

```
mod FLAWED-IFF-INIT is
  pr FLAWED-IFF .
  op init : -> Config .
  var Cs : Soup{Cipher} .
  vars P Q : Prin .
  var K : Key .
  var R : Rand .
  var NW : Soup{Msg} .
  var OCs : Soup{OComp} .
  eq init = {(nw: empty) (prins: (p1 p2 intr)) (rands: (r1 r2))} .
endm
```

4.3.3 Maude の search コマンドによる不変性モデル検査

ここでのモデル検査は侵入者と認証が確立した時、利用された鍵が侵入者とは別グループの鍵だった状態を探索する。実行した結果は下記になる。

```
Maude> search [1] in FLAWED-FIFF-INIT : init =>* {(nw: (cm(P,P,intr,R) rm(intr,intr,
     P,enc(K,R)) NW)) OCs}
> such that (not P == intr) and K == key(P) .
search [1] in FLAWED-FIFF-INIT : init =>* {OCs nw: (NW cm(P, P, intr, R) rm(intr,
     intr, P, enc(K, R)))} such that not P == intr and K == key(P) = true .

Solution 1 (state 4523)
states: 4524  rewrites: 19820 in 47ms cpu (48ms real) (419852 rewrites/second)
OCs --> prins: (intr p1 p2) rands: r2
NW --> cm(intr, intr, p1, r1) rm(p1, p1, intr, enc(gKey1, r1))
P --> p1
R --> r1
K --> gKey1
```

反例が見つかった。NW の状態が cm(intr, intr, p1, r1) rm(p1, p1, intr, enc(gKey1, r1)) となっている。これは intr(侵入者) がネットワークに流れた乱数 R を利用して、p1 へ cm メッセージを送り、p1 から rm メッセージを取得した事を表す。この時点で侵入者と p1 の認証が確立したことになる。しかし、侵入者は p1 とは別の鍵を持つグループ外のメンバーである。誤識別をしたことになる。

4.4 FIFF の修正

FIFF では暗号化を行う際に実行者の識別子を含めずに行った為に敵味方の識別性を満たすことができなかった。その為ここでは暗号化を行う際に下記の修正を行った。

enc(K,R,Q) ... enc(利用する共通鍵, 乱数, 自身の ID)

具体的な Maudeno 修正は下記の CIPHER モジュールになる。

```
fmod CIPHER is
  pr PRIN .
  pr RAND .
  pr KEY .
  sort Cipher .
  op enc : Key Rand Prin -> Cipher .
endfm
```

enc 関数の引数に利用する共通鍵、乱数、自身の ID となる。

4.4.1 Maude の search コマンドによる不変性モデル検査

認証が確立した時、認証したユーザと鍵の組み合わせを探索する。結果は下記になる。

```
Maude> search [1,6] in IFF-INIT : init =>* {(nw: (rm(Q1,Q2,P,enc(K,R,Q2)) NW)) OCs}
      such that not (K =/= gKey2 implies Q2 =/= intr) .
search [1, 6] in IFF-INIT : init =>* {OCs nw: (NW rm(Q1, Q2, P, enc(K, R, Q2)))}
     such that not (K =/= gKey2 implies Q2 =/= intr) = true .

No solution.
states: 2270944  rewrites: 63481869 in 469581ms cpu (473668ms real) (135188
     rewrites/second)
```

深さ 6 においては誤認が成立する状態は存在しなかった。しかし、下記の環境下においては深さ指定をせずに全ての状態空間での探索は現実的な時間内では終わらず、確認できなかった。

- MacBook Pro 16-inch, 2019
- macOS Ventura version13.5
- 2.3 GHz 8-Core Intel Core i9
- 32 GB 2667 MHz DDR4

4.5 状態爆発対策

修正された IFF において深さ指定をせずモデル検査を行うと現実的な時間内では完了が不可能な事がわかった。この状態爆発から改善する為に下記の事を行った。

- Maude のコードに改良を加え、軽量に動くようにした
- JAIST の Large Memory PC Cluster (LMPCC) を活用した

4.5.1 改良した Maude のコード

cm,rm メッセージを送信する際に第一引数の作成者を省略した。重要な部分はメッセージの作成者ではなく、暗号化の仕様である。メッセージの作成者は満たすべき性質つまり、敵味方の識別性とは直接的な関係がなく、省略可能である。利用変数を削減することで変数と変数の組み合わせ数が削減できる為、状態爆発問題の軽減施策になる。変更を加えた内容は下記の通りである。

変更前)
cm(作成者, 送信者, 受信者, 乱数)
rm(作成者, 送信者, 受信者, 暗号文)

変更後)
cm(送信者, 受信者, 乱数)
rm(送信者, 受信者, 暗号文)

4.5.2 LMPCC での実行結果

LMPCC で実行した結果は下記になる。

```
Maude> search [1] in IFF-INIT : init =>* {(nw: (cm(P,intr,R) rm(P,enc(K,R,intr)) NW
      )) OCs} such that (not P == intr) and K == key(P) .
search [1] in IFF-INIT : init =>* {OCs nw: (NW rm(P, enc(K, R, intr)) cm(P, intr, R
      ))} such that not P == intr and K == key(P) = true .

==> PBS: job killed: walltime 604921 exceeded limit 604800
qsub: job 2663547.lmpcc completed
```

結果的に 2 週間全探索し続けたが、完了できず制限時間に引っかかり、強制終了されてしまった。状態爆発を軽減する為に検査とは関連のない変数を省略し、到達可能状態を削減して実行したが、それでも実行完了まではできなかった。

4.5.3 さらに改良した Maude のコード

ここでは乱数の扱いに改良を加える。もともとのコードでは乱数を生成する時は誰から誰へという情報を加えずに生成していたが、今回のコードでは誰から誰に向けて生成した乱数なのかを下記のように定義した。

n(p1,p2,r) ... p1 から p2 向けの乱数を生成したという意 (n は nonce)

これにより cm,rm メッセージ関数は削減できる。nonce の内容から誰から誰への cm メッセージなのかが判別でき、enc の内容から誰から誰へに対する返信メッセージなのかが判別が可能となる。この改良を加えたコードで LMPCC で実行を試みた。しかし、LMPCC で実行した結果は同じく、2 週間では完了できず、強制終了されてしまった。単純な利用変数の軽減だけでは不十分であり、別の観点からのアプローチが必要であると考える。

4.6 別の状態爆発対策

前の項目で利用変数の削減だけでは不十分であることがわかった為、ここでは別の観点での対策を試す。P と Q の 2 者間のメッセージのやり取りを最初にメッセージを送る Initiator と 2 番目にメッセージを送る Responder に分けて Maude で記述するように変更した。

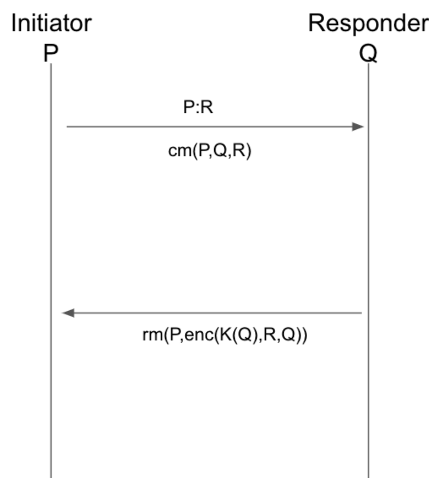


図 4.3: 状態爆発対策を考慮した IFF のメッセージ交換

Maude の変更点は 2 点である。1 点目は Initiator-principal(i-prins) と Responder-principal(r-prins) という観測可能成分を追加する事で Initiator と Responder を分けて記述するようにした点である。下記には最終的な観測可能成分を示す。

観測可能成分	**説明**
(nw: *ms*)	*ms* はネットワークに流れたメッセージのスープである 認証を行う principal のメッセージ交換するネットワークを表す
(i-prins: *ips*)	*ips* は Initiator principal のスープである 最初のメッセージを送る登場人物になる
(r-prins: *rps*)	*rps* は Responder principal のスープである 2 番目にメッセージを送る登場人物になる
(rands: *rs*)	*rs* は乱数のスープである 各 principal が発行する乱数を表す

また、Maude での初期状態を下記のように設定した。

```
1 {(nw: empty) (i-prins: (p1 intr)) (r-prins: (p2 intr)) (rand: (r1 r2))} .
```

2 点目は書き換え規則で if not P == Q の条件を加える事で自分自身にメッセージ送信を行わないようにした点である。

```
crl [sdcm] : {(nw: NW) (i-prins: (P Ps)) (r-prins: (Q Qs)) (rand: (R Rs))}
    => {(nw: (cm(P,Q,R) NW)) (i-prins: (P Ps)) (r-prins: (Q Qs)) (rand: Rs)}
    if not P == Q .

rl [sdrm] : {(nw: (cm(P,Q,R) NW)) OCs}
    => {(nw: (rm(P,enc(key(Q),R,Q)) cm(P,Q,R) NW)) OCs} .

crl [fakecm1] : {(nw: (cm(P2,Q2,R) NW)) (i-prins: (P Ps)) (r-prins: (Q Qs)) OCs
    }
    => {(nw: (cm(P,Q,R) cm(P2,Q2,R) NW)) (i-prins: (P Ps)) (r-prins: (Q Qs))
        OCs}
    if not P == Q .

crl [fakerm1] : {(nw: (cm(P2,Q2,R) NW)) (i-prins: (P Ps)) (r-prins: (Q Qs)) OCs
    }
    => {(nw: (rm(P,enc(key(intr),R,Q)) cm(P2,Q2,R) NW)) (i-prins: (P Ps)) (r-
        prins: (Q Qs)) OCs}
    if not P == Q .

rl [fakerm2] : {(nw: (rm(P2,C) NW)) (i-prins: (P Ps)) OCs}
    => {(nw: (rm(P,C) rm(P2,C) NW)) (i-prins: (P Ps)) OCs} .
```

このことにより Initiator の P と Responder の Q に入る値のパターンが下記のように限定される。

Initiator 変数:P	Responder 変数:Q
p1	p2
p1	intr
intr	p2

表 4.1: 変数 P と変数 Q に入る組み合わせ

自分自身にメッセージを送信し、認証を確立した状態を探索しても同じメンバーであり、味方であることはあらかじめわかっている。敵味方の識別性の性質を調べる上では自分自身にメッセージを送信する事は省略可能である。よって書き換え規則で if not P == Q の条件をくわえる事は調べたい性質には影響が無い事且つ、到達可能状態の削減ができる為、短時間でのモデル検査が期待できる。

4.6.1 Maude の search コマンドによるモデル検査

```
Maude> search [1] in IFF-INIT : init =>* {(nw: (cm(P,intr,R) rm(P,enc(K,R,intr)) NW
    )) OCs} such that (not P == intr) and K == key(P) .
search [1] in IFF-INIT : init =>* {OCs nw: (NW rm(P, enc(K, R, intr)) cm(P, intr, R
    ))} such that not P == intr and K == key(P) = true .
```

```
No solution.
states: 64009  rewrites: 10954395 in 15858ms cpu (15885ms real) (690743 rewrites/
    second)
```

上記の search コマンドによるモデル検査は下記の環境下で行った。モデル検査を実行した結果、数秒で結果が得られた。

- MacBook Pro 16-inch, 2019
- macOS Ventura version13.5
- 2.3 GHz 8-Core Intel Core i9
- 32 GB 2667 MHz DDR4

4.7 状態爆発対策の実験のまとめ

ここでは状態爆発対策の実験まとめを行う。状態爆発の対策を何も実施しなかった 4.4 と対策を実施した 4.6 のコードで扱う到達可能状態数を比較したものが下記である。

Mauden のコード	探索した到達可能状態数
4.4 の修正された IFF コード (但し深さ 6 までの探索)	2270944
4.6 の状態爆発対策のコード	64009

4.4 のコードは LMPCC でもモデル検査が完了できなかった為、深さ指定した場合の探索した到達可能状態数を記している。対策を実施した 4.6 では不要な状態を削減し、効率的に探索できていることがわかる。

4.8 状態爆発対策のまとめ

状態爆発問題を緩和するために下記 2 点を行った。

- 不要な変数の削減
- 不要な変数同士の組み合わせの削減

対策にあたり検査したい性質に影響を与えないものであるかどうかが重要であり、今回の IFF ではうまくいった。しかし、登場人物を増やした場合などは上記の対策だけでは不十分なことも考えられ、3.3.2 で調査をしたようなモデルを複数の小さなモデルに分割し、それらを個別に検査する分割統治のアプローチも必要になってくると考えられる。

第5章　Needham-Schroeder 公開鍵認証プロトコル (NSPK)

NSPK は公開鍵サーバを用いた相互認証プロトコルである。ここでは NSPK を題材にしてどのように状態機械として形式化したか、どのように Maude で記述したか、モデル検査した保証すべき性質の内容、それに対する反例の説明を行う。NSPK は下記の前提条件の元に行われる。

1. プロトコルに参加する各メンバーの公開鍵はプロトコルに参加する他のメンバーに安全に伝達される。
2. 各メンバーの秘密鍵は所有者自身のみが知っているものとする。

NSPK のメッセージの概要図は下記の通りである。

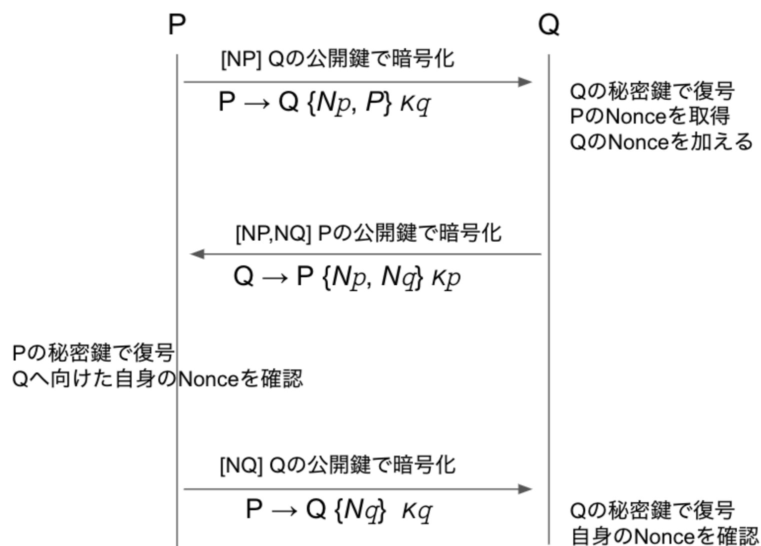


図 5.1: NSPK のメッセージ交換

P と Q はプロトコルに参加するメンバーであり、お互いの公開鍵を所有している。P が Q へ認証を行いたい場合、P はノンスを生成し、P の識別子と共に暗号化

して Q へ送信する。Q は復号化し、P からのノンスを確認し、Q のノンスを生成する。受け取った P のノンスと生成した Q のノンスを暗号化して P へ返信する。P は復号化して自身が送ったノンスを確認し、送られてきた Q のノンスを再度暗号化して Q へ送信する。次により詳細にメッセージの交換部分だけに注目すると下記になる。

$$\text{Challenge : P} \rightarrow \text{Q}\{N_p, P\}_{K_q} \tag{5.1}$$

$$\text{Response : Q} \rightarrow \text{P}\{N_p, N_q\}_{K_p} \tag{5.2}$$

$$\text{Confirmation : P} \rightarrow \text{Q}\{N_q\}_{K_q} \tag{5.3}$$

$_{K_p}$ は p の公開鍵である。N_p は p が生成したノンスである。p が q に送信する為に生成したノンスは n(p, q, r) と表現する。$\{\ \}_{K_p}$ は $_{K_p}$ で暗号化したメッセージである。$_{K_p}$ に対応する秘密鍵を所有するのは本人のみである。Challenge では P は Q へノンスを生成し、P の識別子と共に Q の公開鍵を利用し暗号化して送信する。Response では Q は秘密鍵でメッセージを復号化し、P から Q へ向けたノンスを確認し、P へのノンスを生成する。確認した P から Q へのノンスと生成した P へのノンスを P の公開鍵を利用して暗号化して P へ返信する。Confirmation では P は秘密鍵でメッセージを復号化し、Q の P へ向けたノンスのみを Q の公開鍵を用いて暗号化して再度送信する。Q は秘密鍵で復号化して P へ送信したノンスを確認する。互いに送ったノンスが返却された事を確認して認証が成立する。

5.1 観測可能成分

ここで使う観測可能成分の解説をする。

観測可能成分	説明
(nw: *ms*)	*ms* はネットワークに流れたメッセージのスープである 認証を行う principal のメッセージ交換するネットワークを表す
(prins: *ps*)	*ps* は principal のスープである 認証を行う登場人物になる
(rands: *rs*)	*rs* は乱数のスープである 各 principal が発行する乱数を表す
(nonces: *ns*)	*ns* はノンスのスープである 各 principal が発行するノンスを表す
(cipher: *cs*)	*cs* は暗号文のスープである 各 principal が発行する暗号文を表す

5.2 状態機械としての形式化

ここではどのように状態機械へと落とし込むかについて説明する。状態の集合 S は下記のように定義する。

$S \triangleq$ {
  {(nw: ms) (prins: ps) (rands: rs) (nonces: ns)}
  | $ms \in$ Set(Msg), ps $\in$ Set(Prin), $rs \in$ Set(Rand), $ns \in$ Set(Nonces)
}

Msg、Prin、Rand、Noncesはメッセージ、参加者、乱数、ノンスのソートで、Set(Msg)、Set(Print)、Set(Rand)、Set(Nonces) の集合を意味する。

初期状態の集合 I は下記のように定義する。

$I \triangleq$ {{(nw: empty) (prins: (p1 p2 intr)) (rands: (r1 r2)) (nonces: empty)}}

empty は空集合を意味する。参加者は p1,p2,intr であり、乱数は r1,r2 の 2 つ存在しており、ノンスは空が初期状態である。

状態遷移の集合 T は下記のように定義する。

$T \triangleq$ {(
  {(prins: (P Q Ps)) (rands: (R Rs)) (nw: Ms) (nonces: empty)},
  {(prins: (P Q Ps)) (rands: Rs) (nw: (m(P,P,Q,enc1(Q,n(P,Q,R),P)) Ms)) (nonces: n(P,Q,R))}
) | P,Q $\in$ Prin, R $\in$ Rand, Ps $\in$ PrinSoup, Rs $\in$ RandSoup, Ms $\in$ MsgSoup
} ∪ {(
  {(rands: (R Rs)) (nw: (m($P2$,P,Q,enc1(Q,N,P)) Ms)) (nonces: Ns) OCs},
  {(rands: Rs) (nw:(m(Q,Q,P,enc2(P,N,n(Q,P,R))) m($P2$,P,Q,enc1(Q,N,P)) Ms)) (nonces: (N n(Q,P,R))) OCs}
) | P,$P2$,Q $\in$ Prin, R $\in$ Rand, N $\in$ Nonce, Rs $\in$ RandSoup, Ms $\in$ MsgSoup, Ns $\in$ NonceSoup, OCs $\in$ OComp
} ∪ {(
  {(nw: (m($P2$,Q,P,enc2(P,$N1$,$N2$)) m(P,P,Q,enc1(Q,$N1$,P)) Ms)) (nonces: Ns) (cipher: Cs) OCs},
  {(nw:(m(P,P,Q,enc3(Q,$N2$)) m($P2$,Q,P,enc2(P,$N1$,$N2$)) m(P,P,Q,enc1(Q,$N1$,P)) Ms)) (nonces: ($N2$ Ns)) (cipher: (enc3(Q,$N2$))) OCs}
) | P,$P2$,Q $\in$ Prin, R $\in$ Rand, $N1$,$N2$ $\in$ Nonce, Rs $\in$ RandSoup, Ms $\in$ MsgSoup, Ns $\in$ NonceSoup, Cs $\in$ CipherSoup, , OCs $\in$ OComp
} ∪ {(
  {(prins: (P Q Ps)) (nw: Ms) (nonces: (N Ns)) OCs},
  {(prins: (P Q Ps)) (nw: (m($intr$,P,Q,enc1(Q,N,P)) Ms)) (nonces: (N Ns)) OCs}
) | P,Q,$intr$ $\in$ Prin, N $\in$ Nonce, Ps $\in$ PrinSoup, Ms $\in$ MsgSoup, Ns $\in$ NonceSoup, OCs $\in$ OComp
} ∪ {(
  {(prins: (P Q Ps)) (nw: Ms) (nonces: ($N1$ $N2$ Ns)) OCs},
  {(prins: (P Q Ps)) (nw: (m($intr$,P,Q,enc2(Q,$N1$,$N2$)) Ms)) (nonces: ($N1$ $N2$ Ns)) OCs}
) | P,Q,$intr$ $\in$ Prin, $N1$,$N2$ $\in$ Nonce, Ps $\in$ PrinSoup, Ms $\in$ MsgSoup, Ns $\in$ NonceSoup, OCs $\in$ OComp
} ∪ {(
  {(prins: (P Q Ps)) (nw: Ms) (nonces: (N Ns)) (cipher: Cs) OCs},

$$
\begin{aligned}
&\{(\text{prins: } (P\ \ Q\ \ Ps))\ (\text{nw: } (\text{m}(intr, P, Q, \text{enc3}(Q, N))\ \ Ms))\ (\text{nonces: } (N\ \ Ns))\ (\\
&\quad \text{cipher: } (\text{enc3}(Q, N)\ \text{Cs}))\ \ OCs\}\\
&)\ |\ P, Q, intr \in \text{Prin},\ N \in \text{Nonce},\ Ps \in \text{PrinSoup},\ Ms \in \text{MsgSoup},\ Ns \in \text{NonceSoup},\\
&\quad Cs \in \text{CipherSoup},\ OCs \in \text{OComp}\\
&\} \cup \{(\\
&\ \{(\text{prins: } (P\ \ Q\ \ Ps))\ (\text{nw: } M\ \ Ms)\ \ OCs\},\\
&\ \{(\text{prins: } (P\ \ Q\ \ Ps))\ (\text{nw: } (\text{m}(intr, P, Q, \text{b}(M))\ \ M\ \ Ms))\ \ OCs\}\\
&)\ |\ P, Q, intr \in \text{Prin},\ M \in \text{Msg},\ Ps \in \text{PrinSoup},\ Ms \in \text{MsgSoup},\ OCs \in \text{OComp}\\
&\}
\end{aligned}
$$

最初の遷移ではPからQへのChallengeメッセージ送信を意味する。二番目の遷移ではQからPへのResponseメッセージ送信を意味する。三番目の遷移ではPからQへのConfirmationメッセージ送信を意味する。四番目の遷移ではネットワークに流れたノンスを利用してintrがChallengeメッセージの偽装送信を意味する。五番目の遷移ではネットワークに流れたノンスを利用してintrがResponseメッセージの偽装送信を意味する。六番目の遷移ではネットワークに流れたノンスを利用してintrがConfirmationメッセージの偽装送信を意味する。七番目の遷移ではネットワークに流れた暗号文を利用してintrから他の人へ偽装送信を意味する

5.3 Maudeによる形式仕様

形式化されたものをどのようにMaudeで記述するかについて説明し、Maudeのsearchコマンドによるモデル検査を行う。Maudeのコードで記述したものが下記になる。ここでは主要な部分を抜粋して説明する。下記は初期状態である。

```
1 {(nw: empty) (rand: (r1 r2)) (nonces: empty) (cipher: empty) (prins: (p1 p2
     intr))}
```

下記は書き換え規則である。

```
 1   rl [Challenge] :
 2     {(prins: (P1 P2 Ps)) (rand: (R Rs)) (nw: NW) (nonces: Ns) OCs}
 3     => {(prins: (P1 P2 Ps)) (rand: Rs)
 4     (nw: (m(P1,P1,P2,enc1(P2,n(P1,P2,R),P1)) NW))
 5     (nonces: (if P2 == intr then n(P1,P2,R) Ns else Ns fi)) OCs} .
 6   rl [Response] :
 7     {(rand: (R Rs)) (nw: (m(P3,P1,P2,enc1(P2,N,P1)) NW)) (nonces: Ns) OCs}
 8     => {(rand: Rs)
 9     (nw: (m(P2,P2,P1,enc2(P1,N,n(P2,P1,R))) m(P3,P1,P2,enc1(P2,N,P1)) NW))
10     (nonces: (if P1 == intr then N n(P2,P1,R) Ns else Ns fi)) OCs} .
11   rl [Confirmation] :
12     {(nw: (m(P3,P2,P1,enc2(P1,N1,N2)) m(P1,P1,P2,enc1(P2,N1,P1)) NW)) (nonces:
         Ns) (cipher: Cs) OCs}
13     => {(nw: (m(P1,P1,P2,enc3(P2,N2)) m(P3,P2,P1,enc2(P1,N1,N2)) m(P1,P1,P2,
         enc1(P2,N1,P1)) NW))
14     (nonces: (if P2 == intr then N2 Ns else Ns fi)) (cipher: (enc3(P2,N2) Cs))
         OCs} .
15   rl [fake1] :
```

```
16      {(prins: (P1 P2 Ps)) (nw: NW) (nonces: (N Ns)) OCs}
17      => {(prins: (P1 P2 Ps)) (nw: (m(intr,P1,P2,enc1(P2,N,P1)) NW)) (nonces: (N
            Ns)) OCs} .
18    rl [fake2] :
19      {(prins: (P1 P2 Ps)) (nw: NW) (nonces: (N1 N2 Ns)) OCs}
20      => {(prins: (P1 P2 Ps)) (nw: (m(intr,P1,P2,enc2(P2,N1,N2)) NW)) (nonces: (
            N1 N2 Ns)) OCs} .
21    rl [fake3] :
22      {(prins: (P1 P2 Ps)) (nw: NW) (nonces: (N Ns)) (cipher: Cs) OCs}
23      => {(prins: (P1 P2 Ps)) (nw: (m(intr,P1,P2,enc3(P2,N)) NW)) (nonces: (N Ns)
            ) (cipher: (enc3(P2,N) Cs)) OCs} .
24    rl [fake4] :
25      {(prins: (P1 P2 Ps)) (nw: (M NW)) OCs}
26      => {(prins: (P1 P2 Ps))  (nw: (m(intr,P1,P2,b(M)) M NW)) OCs} .
```

上記の書き換え規則に出てくるP1、P2、P3はPrinの変数であり、p1もしくはp2もしくはintrのいずれかを意味する。書き換え規則Challengeは(5.1)のChallengeに相当し、参加者(p1,p2,intr)の間でのChallengeメッセージ送信を意味する。書き換え規則Responseは(5.2)のResponseに相当し、参加者(p1,p2,intr)の間でのResponseメッセージ送信を意味する。書き換え規則Confirmationは(5.3)のConfirmationに相当し、参加者(p1,p2,intr)の間でのConfirmationメッセージ送信を意味する。書き換え規則fake1ではChallengeメッセージの偽装に相当し、ネットワークに流れたノンス1つを利用してintrが自身を含めた他者へ送りつけている事を意味している。書き換え規則fake2ではResponseメッセージの偽装に相当し、ネットワークに流れたノンス2つを利用してintrが自身を含めた他者へ送りつけている事を意味している。書き換え規則fake3はConfirmationメッセージの偽装に相当し、ネットワークに流れたノンス1つを利用してintrが自身を含めた他者へ送りつけている事を意味している。書き換え規則fake4はネットワークに流れた暗号文を利用してintrが自身を含めた他者へ送りつけている事を意味している。ここでの暗号文は書き換え規則Challenge、Response、Confirmation、fake1、fake2、fake3、fakeのいずれかで作られた暗号文になる。

ここででてくる関数の意味は下記の通りである。

- m(P,P,Q,C)
  - メッセージ送信を表す
  - 第一引数のPは作者。第二引数のPは送信者。第三引数Qは宛先。第四引数は暗号文を表す
- n(P,Q,R)
  - ノンスを表す
  - 第一引数Pは作者。第二引数Qは宛先。第三引数は乱数を表す
- enc1(Q,N,P)

– 第一メッセージの暗号化を表す
– 第一引数の Q は宛先。第二引数 N はノンス。第三引数 P は作成者を表す

- enc2(P,N,$N1$)
  – 第一メッセージのリプライの暗号化を表す
  – 第一引数の P は宛先。第二引数 N は第一メッセージのノンス。第三引数 N1 は新たに作成したノンスを表す

- enc3(Q,$N1$)
  – リプライに対する確認メッセージの暗号化を表す
  – 第一引数の Q は宛先。第二引数 N1 はリプライメッセージで作成したノンスを表す

書き換え規則 Challenge,Response,Confirmation においてメッセージの宛先が intr の時だけ nonces の観測可能成分にノンスを追加している。またこの時ノンス自体も intr 宛に作られたものと仮定する。そして後の書き換え規則 fake1、fake2、fake3、fake 4 では intr が収集したノンスまたは暗号文を使い、他者へ偽装送信を意味している。

5.4 Maude の search コマンドによる不変性モデル検査

ここでは観測可能成分の nonces 項目に着目する。nonces 項目は intr 宛のメッセージの中にあった intr 向けに作られたノンスが存在するはずである。しかし、宛先も作者も intr ではないノンスが存在した時はノンスが関係のない他者へ漏えいした事になり、悪用される可能性がある。その為ノンスの作者、宛先が共に intr ではないものを探索した。

```
Maude> search [1] in NSPK : init =>* {(nonces: (n(P1,P2,R) Ns)) OCs} such that P1
    =/= intr and P2 =/= intr .
search [1] in NSPK : init =>* {OCs nonces: (Ns n(P1, P2, R))} such that P1 =/= intr
     and P2 =/= intr = true .

Solution 1 (state 91853)
states: 91854  rewrites: 940196 in 1803ms cpu (1810ms real) (521275 rewrites/second
    )
OCs --> nw: (m(p1, p1, intr, enc3(intr, n(p2, p1, r2))) m(p1, p1, intr, enc1(intr,
    n(p1, intr, r1), p1)) m(p2, p2, p1, enc2(p1, n(p1, intr, r1), n(p2,
    p1, r2))) m(intr, p1, p2, enc1(p2, n(p1, intr, r1), p1)) m(intr, intr, p1, enc2
        (p1, n(p1, intr, r1), n(p2, p1, r2)))) rand: empty cipher: enc3(
    intr, n(p2, p1, r2)) prins: (p1 p2 intr)
Ns --> n(p1, intr, r1)
```

P1 --> p2
P2 --> p1
R --> r2

反例が 1 つ出力された。その時の nw 項目と show path 91853 の解析結果を要約すると下記のようなメッセージ送受信があり、なりすましが成立していたことがわかる。ノンスの秘匿性が保たれていない状態を探すことで、それを悪用したなりすましが成立した事を確認できた。

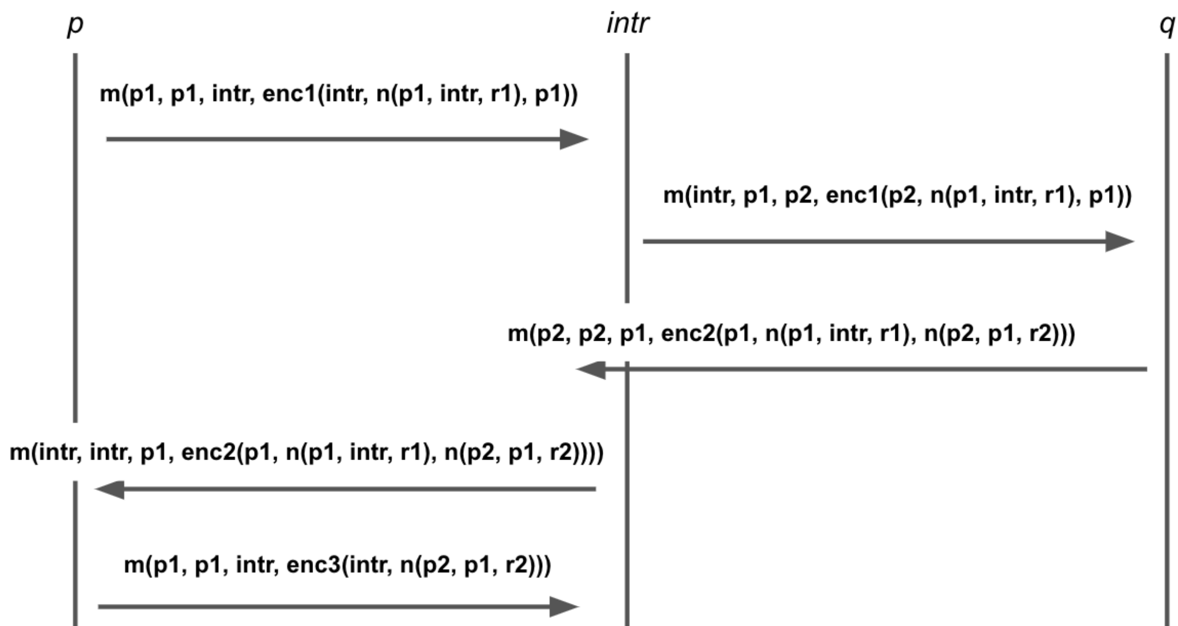


図 5.2: なりすましが成立したメッセージ交換

第6章　Needham‐Schroeder‐Lowe 公開鍵認証プロトコル (NSLPK)

Gavin Lowe[5] によって NSPK の問題点を指摘し同時に修正版の NSLPK も記述された。NSLPK のメッセージ交換に着目する。NSPK の時と同様にメッセージ交換を抜粋すると下記になる。

$$\text{Challenge : P} \rightarrow \text{Q}\{N_p, P\}_{K_q}$$

$$\text{Response : Q} \rightarrow \text{P}\{N_p, N_q, Q\}_{K_p}$$

$$\text{Confirmation : P} \rightarrow \text{Q}\{N_q\}_{K_q}$$

NSPK と NSLPK の違いは Response メッセージを作成する際に Q の ID を含めている点である。その為、誰からの Response メッセージかが判別可能になり、なりすましを防ぐことが可能になる。

6.1　Maude による形式仕様

NSPK と異なる点は Response メッセージの暗号化を行う enc2 が下記のように変更になる。

変更前) enc2(宛先, ノンス, ノンス)
変更後) enc2(宛先, ノンス, ノンス, 自身の ID)

6.2　Maude の search コマンドによる不変性モデル検査

5.4 で実行した search コマンドをここでも実行する。実行する search コマンドは下記になる。

```
search [1] in NSLPK : init =>* {(nonces: (n(P1,P2,R) Ns)) OCs} such that P1 =/=
    intr and P2 =/= intr .
```

NSPK の時と同様に観測可能成分の nonces 項目に着目する。intr 宛のメッセージの中にあった intr 向けのノンスが nonces 項目に存在する想定である。よってノンスの作者、宛先が intr 以外のものがある場合はノンスが関係のない者に漏えいしたことになる。その状態を探索する。モデル検査の結果は現実的な時間内では完了はできなかった。状態爆発が発生し、探索が完了しなかった。

6.3 状態爆発対策

IFF の 4.6 で取った対策をここでも活用する。変更点は主に 3 点ある。1 点目は P と Q の 2 者間のメッセージのやり取りを最初にメッセージを送る Initiator と 2 番目にメッセージを送る Responder に分けて Maude で記述するように変更した。ここで使う観測可能成分を下記に記す。

観測可能成分	**説明**
(nw: *nw*)	*nw* はネットワークに流れたメッセージのスープである 認証を行う principal のメッセージ交換するネットワークを表す
(i-prins: *ips*)	*ips* は Initiator principal のスープである 最初のメッセージを送る登場人物になる
(r-prins: *rps*)	*rps* は Responder principal のスープである 2 番目にメッセージを送る登場人物になる
(rands: *rs*)	*rs* は乱数のスープである 各 principal が発行する乱数を表す
(nonces: *ns*)	*ns* はノンスのスープである 各 principal が発行するノンスを表す

2 点目は暗号化のパターンから誰から誰に送信したメッセージなのかまとめて表現するようにした。下記にまとめる。暗号化の形式がそれぞれ異なる為、それぞれの異なるメッセージ表現も可能になった。このことにより、メッセージ送信を表現していた m 関数が削減が可能である。

- c1(Q,N,P)
  - NSPK の enc1 に相当
  - 最初の Challenge メッセージの意味も兼ねる
  - 第一引数の Q は宛先。N は Q 向けのノンス。P は送信者を表す
- c2(P,N,N',Q)

  - NSPK の enc2 に相当
  - Response メッセージの意味も兼ねる
  - P は宛先。N は Challenge メッセージの中に存在したノンス。N' は Response で作成した P 向けのノンス。Q は送信者を表す

- c3(Q,N')
  - NSPK の enc3 に相当
  - Confirmation メッセージの意味も兼ねる
  - Q は宛先。N' は Response メッセージの中に存在したノンスを表す

3 点目は書き換え規則の中で if not P == Q の条件を加える事で自分自身にメッセージ送信を行わないようにした点である。4.6 で記載している内容と同じ理由である。

```
1   crl [challenge] :
2     {(nw: NW) (nonces: Ns) (rand: (R Rs)) (i-prins: (P Ps)) (r-prins: (Q Qs))}
3     =>
4     {(nw: (c1(Q,n(P,Q,R),P) NW))
5       (nonces: (if Q == intrdr then (n(P,Q,R) Ns) else Ns fi))
6       (rand: Rs) (i-prins: (P Ps)) (r-prins: (Q Qs))}
7   if P =/= Q .
8
9   rl [Response] :
10    {(nw: (c1(Q,N,P) NW))
11      (rand: (R Rs)) (nonces: Ns) OCs}
12    =>
13    {(nw: (c2(P,N,n(Q,P,R),Q) c1(Q,N,P) NW))
14      (rand: Rs)
15      (nonces: (if P == intrdr then (N n(Q,P,R) Ns) else Ns fi))
16      OCs} .
17
18  rl [Confirmation] :
19    {(nw: (c2(P,N,N',Q) c1(Q,N,P) NW))
20      (nonces: Ns) OCs}
21    =>
22    {(nw: (c3(Q,N') c2(P,N,N',Q) c1(Q,N,P) NW))
23      (nonces: (if Q == intrdr then (N' Ns) else Ns fi))
24      OCs} .
25
26  crl [fake1] :
27    {(nw: NW) (nonces: (N Ns)) (i-prins: (P Ps)) (r-prins: (Q Qs)) OCs}
28    =>
29    {(nw: (c1(Q,N,P) NW))
30      (nonces: (N Ns)) (i-prins: (P Ps)) (r-prins: (Q Qs)) OCs}
31  if P =/= Q /\ Q =/= intrdr .
32
33  crl [fake2] :
34    {(nw: NW) (nonces: (N N' Ns)) (i-prins: (P Ps)) (r-prins: (Q Qs)) OCs}
```

```
35      =>
36      {(nw: (c2(P,N,N',Q) NW))
37        (nonces: (N N' Ns)) (i-prins: (P Ps)) (r-prins: (Q Qs)) OCs}
38    if P =/= Q /\ P =/= intrdr .
39
40    crl [fake3] :
41      {(nw: NW) (nonces: (N Ns)) (i-prins: (P Ps)) (r-prins: (Q Qs)) OCs}
42      =>
43      {(nw: (c3(Q,N) NW))
44        (i-prins: (P Ps)) (r-prins: (Q Qs)) OCs}
45     if Q =/= intrdr .
```

6.4 Maudeのsearchコマンドによる不変性モデル検査

ここではNSPKの時と同様に観測可能成分のnonces項目に着目する。Challenge,Response,Confirmationメッセージにおいてメッセージの宛先がintrの時だけnoncesの観測可能成分にノンスを追加されている。よってnonces項目にあるノンスはintr向けに作成されたものになる。しかし、intr向けに作られていないノンスが存在したとすると関係のない他者へ漏えいした事になり、悪用される可能性がある。その為ノンスの作者、宛先が共にintrではないものを探索し、機密性が保たれているかを確かめる。

```
Maude> search [1] in NSLPK : init =>* {(nonces: (n(P,Q,R) Ns)) OCs} such that P =/=
      intrdr /\ Q =/= intrdr .
search [1] in NSLPK : init =>* {OCs nonces: (Ns n(P, Q, R))} such that P =/= intrdr
      = true /\ Q =/= intrdr = true .

No solution.
states: 6941  rewrites: 191137 in 288ms cpu (289ms real) (662655 rewrites/second)
```

反例もなく、探索が無事に完了することがわかった。

第7章　おわりに

ここではさまざまな事例を通して Maude を用いた形式仕様作成とモデル検査を行ってきたまとめを報告する。

7.1　まとめ

本課題研究では Test&Set、Qlock、IFF、NSPK、NSLPK を題材にして Maude を用いた形式仕様作成とモデル検査を実施して課題研究としてまとめた。それぞれの題材の中であえて誤りを注入する事でより深い理解に努めた。Qlock では状態が無限に作り出される現象に遭遇したが、幅優先探索を採用している Maude の search コマンドを活用することにより回避した。またプロセス数を増やしていった場合などの実験も行い、状態爆発になる現象とそれ対する既存研究について調査を行った。

IFF ではモデル検査の課題の 1 つでもある状態爆発に遭遇し、Maude で利用する変数の数の削減を行い、大容量メモリ計算機 (lmpcc) での実行を 2 週間試みたが、回避できなかった。さらに検査したい性質に関係のない変数を削減したり、変数の組み合わせを効率的に行えるよう Maude のコードを工夫することで大容量メモリ計算機 (lmpcc) でなくとも実行可能なように回避した。改良を加えたコードでの到達可能状態数も大きく削減できた。

NSPK では IFF で得られた知見を活用し、公開鍵、ノンスを使ったプロトコルに対して Maude を用いた形式仕様作成とモデル検査を実施した。ノンスの秘匿性が保たれているかをモデル検査した。NSLPK でも同様に IFF で得られた知見を活用し、検査したい性質に関係のない変数を削減、変数の組み合わせの効率化を実施し、状態爆発を回避しながらノンスの秘匿性が保たれているかをモデル検査した。今回の調査により得られた知見として下記があげられる。

1. Qlock においてプロセス識別子の待ち行列への操作を不可分にしない場合、到達可能状態が無限であることの理由を明らかにしたこと

2. 認証プロトコルのモデル検査で遭遇した状態爆発問題をプロトコルの参加者をイニシエーターとレスポンダーに明確に分けることで対応可能であることを示したこと

また今回の研究課題報告書を通して下記2つの課題と対策を考察する。

第一に状態空間の爆発問題である。到達可能状態を全て探索する必要がある為、どうしても時間とモデル検査の実行性能を求められる。近年ではクラウドサービスで高性能なクラスタなどが比較的簡単に利用できるようになってきているが、普及には至っていない。また到達可能状態の削減も必要である。削減する為にはコードを修正する必要があるが、検証対象のシステムの性質、モデリングの理解度に依存してしまう。

この課題に対してはモデリングの理解度を高める為には人が理解しやすいようにする為の可視化可能なツールを活用する必要がある。既存技術として SMGA（State Machine Graphical Animation）がある。SMGA は状態機械の状態遷移を視覚的に表現する技術である。SMGA を活用した研究事例が存在する [6]。これは Mellor-Crummey と Scott によって提案された相互排他プロトコル（MCS プロトコル）[7] が相互排他性を享受することの証明に活用されている。また別の事例としてシステムやプロトコルが望ましい性質を満たしているかを証明する為のレンマ（補題）を推測する為に活用している事例がある [8]。SMGA では観測可能な構成要素に対してテキスト表示と視覚表示する。SMGA が補助的な情報を提供しこの情報を使い人間がレンマを推測する。SMGA が支援をし、研究を促進している例である。状態機械を視覚的に捉える事ができればモデリングの理解に大きく役立つと言える。また最初から大きなモデルを作らないという工夫が必要である。少しでも不具合がないシステムを作ろうとすると必然的にモデル検査する対象が大きくなり、モデルも大きくなる。保証したい性質を絞り、可能であれば分けてモデリングして実行する事が必要である。

第二にモデル検査を導入するコストの問題である。これまでにもモデル検査の有効性と適用提案は数多く存在するが、実際のシステム開発現場でモデル検査を利用しているケースは多くない。理由は導入コストがかかり、導入コストに見合うだけのシステムであるかという判断が存在する。IPA 独立行政法人 情報処理推進機構から出ている「情報系の実稼働システムを対象とした形式手法適用実験報告書」[9] の 7.6 では検証対象のシステムのドメイン知識が必要でドメイン知識を持つ開発者が仕様記述言語も記述するか、もしくはドメイン知識を持つ開発者と共同で作業を行わないと無駄な作業が発生するとある。両方の知見をもつ開発者が必要であり、結果的にコストがかかる。また仕様記述言語の知見を有するエンジニアを確保または習得する時間的なコストも存在する。システム開発で利用するプログラミング言語とは別に仕様記述言語の扱いを習得する必要がある。一般的にシステム開発で利用するプログラミング言語の扱えるエンジニアと比べ仕様記述言語を扱えるエンジニアは圧倒的に少ない。この点についても課題であると考える。

この課題に対してはビジネス的な判断も関わる為難しいポイントであるが、うまくモデル化さえできれば不具合のシナリオを事前に考える必要はなく検査が可能であり、反例が出力された時に読み解くことで十分になる。このことにより検

査対象のシステムの性質の表現に注力でき、システムの理解が深まる副次的な効果も期待できる。この副次的な効果も考慮に入れて検討すると良いと考える。

7.2 今後の課題

この課題研究報告書では活性（liveness）の性質に関しての検査は取り扱っていない。活性のモデル検査とはシステムが最終的にある状態に到達することができるかを保証する検査である。また検証を行うには線形時相論理（LTL）の論理式を取り扱う必要がある。Maude は書き換え規則によって作られる状態を探索する search コマンドに加えて線形時相論理モデル検査 [10] も可能である。活性の性質を検査する事は主に下記の場合において効果的である。

- システムが常に何かの処理を実行し続けている事を保証する場合
- システムが無限ループになっていない事を保証する場合

この課題研究報告書でも取り扱った test & set を例にして活性の性質の検証を考えると複数のプロセスが相互排除的に共用資源を利用したい場合、あるプロセスはいつかは必ず利用することができるという性質を検証することが可能である。同じく Qlock を例にして活性の性質の検証を考えると複数のプロセスが相互排除的に共用資源を利用したい場合、あるプロセスがクリティカルセクションに入ろうとしている時、そのプロセスが最終的にクリティカルセクションに入ることができるという性質を検証することが可能である。LTL を考える上でクリプケ構造 [11] の活用も必要になる。2.1 で扱った状態機械に原子命題の集合 P とラベル関数 L を追加したもので状態機械を拡張した構造と定義することが可能である。またクリプケ構造は LTL の意味論を表現するために用いられ、今後の学修課題の一つである。

本課題研究報告書では状態爆発に対して緩和する手段として利用変数の削減と変数の組み合わせの効率化を行ったが、プロセスなどの数を増やすとそのうち状態爆発が発生する。それを改善するための方法を既存研究を踏まえながら習得していく必要がある。既存研究の一例として分割統治を行うアプローチがある。分割統治によるモデル検査は到達可能状態空間を複数の部分空間に分割し、分割により得られる部分空間をモデル検査の対象とすることで状態空間爆発問題を緩和することである。分割により得られる部分空間を並列にモデル検査することで状態爆発の抑制と共に検査時間を短縮が可能である。不変性のモデル検査の場合は分割するだけだが、活性など他の性質の場合は分割だけでは十分ではなく、部分空間に対して何を検査すればよいのかについて工夫が必要である。これら視点での既存研究を下記 2 つ報告する。

7.2.1 Large-Scale Directed Model Checking LTL

この研究 [12] では大規模なシステムの LTL モデル検査を効率的に行う為の手法を提案している。提案の概要を下記にまとめると

- モデルを分割し状態を外部メモリに保存することで探索を効率化する
- 次に複数の処理機を使って並列に探索を行う
- 反例が得られる可能性が高いものを優先的に探索をすることで反例が存在する場合は最短で出力できるようにしている

7.2.2 A Divide & Conquer Approach to Leads-to Model Checking

Leads-to プロパティとはシステムあるいはプロトコルがある状態になったらそのうち必ず他の状態になるといった性質である。この研究 [2] ではただ分割するだけでは不十分であり、部分空間に対して何を検査すればよいのかについて工夫を必要とする。元の無限数列に対する Leads-to のモデル検査問題は分割された複数の無限数列に対する lead-to モデル検査問題と等価であることを証明し、状態空間爆発問題を緩和を試みている。ケーススタディとしてこの課題研究報告書でも取り扱った Qlock と MCS(A Mutual Exclusion Protocol)[7] を扱っている。

謝辞

本課題研究報告書を執筆するにあたり多くのお力添えをいただきました。主指導教員の緒方和博教授には粘り強く、熱意を持ってご指導頂きました。緒方先生のお力添えがなければ、執筆できませんでした。深く心より感謝いたします。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時間をいただき、多くのアドバイスをいただきました。ありがとうございました。最後に学生生活を支えてくれた家族に感謝します。ありがとうございました。

参考文献

[1] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, Rubén Rubio, and Carolyn Talcott. Maude manual (version 3.0). SRI International, 2020.

[2] Yati Phyo, Canh Minh Do, and Kazuhiro Ogata. A Divide & Conquer Approach to Leads-to Model Checking. The Computer Journal, Vol. 65, No. 6, pp. 1353–1364, 02 2021.

[3] Canh Minh Do, Yati Phyo, Adrián Riesco, and Kazuhiro Ogata. Optimization techniques for model checking leads-to properties in a stratified way. ACM Trans. Softw. Eng. Methodol., Vol. 32, No. 6, sep 2023.

[4] Canh Minh Do, Yati Phyo, and Kazuhiro Ogata. Sequential and parallel tools for model checking conditional stable properties in a layered way. IEEE Access, Vol. 10, pp. 133749–133765, 2022.

[5] Gavin Lowe. An attack on the needham- schroeder public- key authentication protocol. Information processing letters, Vol. 56, No. 3, 1995.

[6] Dang Duy Bui and Kazuhiro Ogata. Better state pictures facilitating state machine characteristic conjecture. Multimedia Tools and Applications, Vol. 81, No. 1, pp. 237–272, 2022.

[7] John M Mellor-Crummey and Michael L Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems (TOCS), Vol. 9, No. 1, pp. 21–65, 1991.

[8] Dang Duy Bui, Duong Dinh Tran, Kazuhiro Ogata, and Adrian Riesco. Integration of state machine graphical animation and maude to facilitate characteristic conjecture: an approach to lemma discovery in theorem proving. Multimedia Tools and Applications, pp. 1–34, 2023.

[9] 独立行政法人情報処理推進機構 技術本部ソフトウェア・エンジニアリング・センター統合系システム・ソフトウェア信頼性基盤整備推進委員

会.　情報系の実稼働システムを対象とした形式手法適用実験報告書. https://www.ipa.go.jp/archive/files/000004621.pdf.

[10] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The maude ltl model checker. Electronic Notes in Theoretical Computer Science, Vol. 71, pp. 162–187, 2004.

[11] Saul A. Kripke. A formal system for semantics of modal logic. Journal of Symbolic Logic, Vol. 24, No. 4, pp. 323–334, 1959.

[12] Stefan Edelkamp and Shahid Jabbar. Large-scale directed model checking ltl. In International SPIN Workshop on Model Checking of Software, pp. 1–18. Springer, 2006.