修 士 論 文

演繹的検証法と探索的検証法の組み合わせについての研究

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

川崎 恵久

2009 年 3 月

修 士 論 文

演繹的検証法と探索的検証法の組み合わせについての研究

指導教官　二木厚吉 教授

審査委員主査　二木厚吉 教授
審査委員　Rene VESTERGAARD 准教授
審査委員　緒方和博 准教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

0610027川崎 恵久

提出年月: 2009 年 2 月

概 要

近年，ソフトウェアや情報システムにおいて，高信頼性や安全性を重要視するようになり，それらの性質を十分に検証する必要がある．現在，ソフトウェア・システムにおける検証方法は大きく分けて二つある．一つはモデル検査を用いた探索的な検証法で，もう一つは推論ベースの演繹的な検証法である．二つの検証法にはそれぞれ異なる特徴やメリット・デメリットを持っている．例えば，無限の状態をとるシステムをモデル検査する場合には，何らかの方法で有限状態に抽象化しなければならない．しかし，推論による検証は，無限の状態でも検証が可能である，一方では，推論による検証は，人の手による作業がモデル検査よりも必要である，などが考えられる．

そこで本研究では，演繹的な検証法と探索的な検証法の良いところを上手く組み合わせ，従来の検証法よりもより効率的な検証法を提案することを目的とする．

無限の状態空間をとるシステムを検証したいと考えた場合，演繹的な検証を用いることが出来るが，人間の手による労力が多くかかってしまう．そこで，モデル検査を用いればより速く検証できるのではないかと考える．しかし，モデル検査は無限状態は扱えず，何らかの方法で有限状態に落とし込まなければならない，また，無限状態であっても状態が大きすぎると検証できないという欠点がある．

何らかの状態の抽象化・有限化手法をとり，状態を有限化できた場合，それらの方法がシステムや仕様において本当に成立しているのかという，正当性の証明が必要である．

代数仕様記述言語 CafeOBJ は，順序ソート項書換理論に基づいた代数仕様言語であり，等式を書換規則として解釈し，計算機上で実行することが可能である．また，CafeOBJ には演繹的な検証法と探索的な検証法に相当する二つの検証法を持つ．それぞれ証明譜による検証，Search コマンドを用いたモデル検査による検証がそれに当たる．CafeOBJ は OTS（観測遷移システム）に基づいて記述することが出来る．CafeOBJ/OTS 法に基づいて記述された仕様は，証明譜の検証が可能で，また，Search コマンドによる検証が可能である．それにより，両検証間を同時に扱う場合，仕様の変換や仕様が同一であることを示すための証明などの必要がない．

CafeOBJ において Search コマンドを用いて全探索のモデル検査を実行する場合，withStateEq と観測等価関数を用いて全探索する方法がある．また，もう一つの方法として，等式によって状態を有限化・抽象化する方法がある．withStateEq と観測等価関数によるモデル検査は，ある二つの状態において観測等価関数に基づき探索枝が一致したと判断された場合，その状態を等価とみなし，状態の削減を行いながら探索する方法である．等式によって状態を有限化・抽象化する手法は，証明譜により正当性が証明された状態の抽象化を意味する等式を宣言してあらかじめ状態を縮退させ，モデル検査にかける方法である．後者の方法は，演繹的な検証と探索的な検証の組み合わせと考えられる．

以上の方法に対し相互排除プロトコル QLOCK で検証の有効性を示している．QLOCK においては，等式を用いる検証法は，withStateEq と観測等価関数を用いる検証よりも探索時間が短いことが示されている．

しかし，他のプロトコルに対し有効性が確かめられていないため，QLOCK とは性質の異なるプロトコルに対し有効性を求める必要があると考えられる．等式を用いた状態の有限化・抽象化する手法に関して，等式の発見の方法や，等式の分類などの分析が行われていない．そこで本研究は以下のことを行う．

- 通信プロトコル SCP に対して検証法を適用する
- 等式の発見法や分類の体系化

QLOCK とは異なる性質を持つプロトコルに対して検証を行うことで，有効性を高め，等式の発見や分類の分析を行い体系化することで，検証の効率化を目指す．

目 次

第1章　はじめに

本章では，序論として本研究の背景と目的，本稿の構成について述べる．

1.1　背景

近年，ソフトウェア・システムにおける検証方法は大きく分けて二つある．一つはモデル検査を用いた探索的な検証法で，もう一つは推論ベースの演繹的な検証法である．二つの検証法にはそれぞれ異なる特徴を持っている．

無限の状態空間を持つシステムをモデル検査を用いて検証する場合，時間的な問題で解が出ない場合がある．そういったシステムをモデル検査によって検証するには，無限の状態を有限の状態に落とし込む必要がある．一方，推論ベースの演繹的な検証法は，有限の状態に落とさずとも，無限の状態をとるシステムをそのまま検証出来る．しかし，推論ベースの検証法は，モデル検査による検証に比べて，人の手による作業が多くかかるという欠点がある．

こういったように，モデル検査と推論ベースの間には検証の性質の違いがあり，また，二つの検証法の間にはメリット・デメリットも存在する．そこで，演繹的な検証法と探索的な検証法を上手く組み合わせることができれば，従来の片方の検証法よりもより効率的な検証法が出来ると考えられる．

1.2　目的

本研究の目的は，演繹的な検証法と探索的な検証法を上手く組み合わせることにより，従来の検証法よりも効率的な検証方法を提案し，その検証法の効果を示すことである．

現在，代数仕様記述言語 CafeOBJ[1][2] には演繹的な検証法と探索的な検証法に相当する二つの検証法を持つ．それぞれ証明譜による検証 [3]，Search コマンドを用いたモデル検査による検証がそれに当たる．CafeOBJ/OTS 法によって記述された仕様は，証明譜の検証が可能で，また，CafeOBJ/OTS 法の上で Search コマンドによる検証が可能である．それにより，両検証間における，仕様の変換や仕様が同一であることを示すための証明などの必要がない．

類似の研究として，モデル検査を用いたシステム検証の支援に関する研究 [4] 及び，OTS/CafeOBJ から OTS/Maude への仕様変換の研究 [5] がある．前者は SMV[6] と CafeOBJ

の組み合わせを目指し，後者はMaude[7]とCafeOBJの組み合わせを示している．しかしどちらも，異なる言語を使用しているため，仕様間の関係性を証明しなくてはならないため，複雑な変換や証明が必要になっている．

そこで，CafeOBJの検証機能を用いて，Searchコマンドを用いたモデル検査の検証における状態の有限化・抽象化する手法において，それらの有限化・抽象化手法における正当性を証明譜による検証で行うという手法を提案する．これは演繹的な検証法と探索的な検証法のそれぞれの特徴の有効活用を実現している．

検証法の効果を示すことについては，性質の異なる二つのプロトコルに対しこれらの検証方法を試し，その効果を考察する．また，組み合わせた検証法の一般化・体系化についても論じる．

1.3 本稿の構成

本稿の構成としては，まず代数仕様記述言語CafeOBJが持つ検証法について説明し，次にそれらを用いた検証法を，相互排除プロトコルを用いて説明する．QLOCKとは性質のことなる通信プロトコルを用いて実験を行い，この検証法の特徴や性質を論じる．

第2章　検証手法

本章では，具体的な演繹的検証法と探索的検証法の手法を述べ，それらの特徴・違いを論じる．また，それらを実現する形式仕様記述言語に関して説明する．

2.1　形式手法

形式手法（Formal Method）[9] とは，数理論理学等に基づき品質の高いソフトウェア・システムを効率よく開発するための手法の総称である．

ソフトウェア開発において，下流工程における誤りは，仕様・設計から見直さなければならない可能性があり，多大なコストがかかる．そのため，開発の効率を高めるには，上流工程において，早期に誤りを発見・分析し，ソフトウェアを開発する必要がある．

形式手法の特徴として，システムの満たすべき性質を正確に，厳密にモデルで表現することができる．これによりシステムへに関する理解を明確にすると共に，システムの満たすべき性質について科学的・系統的な分析や検証が可能になる．

その結果として，非曖昧性を高めたり誤りを早期に発見し手戻りを防いだり，分析・検証によりソフトウェア・システムの品質を高めたりすることができる．よって，形式手法は，高度な信頼性や安全性を求められるソフトウェア・システムにおいては特に重要であり，近年注目されている．

形式手法を用いて検証するためには，形式仕様記述言語を用いて形式仕様という形で表現する必要がある．形式仕様記述言語とは，意味や構造を曖昧性の無い数学モデルや論理体系に基づいて定義された仕様を記述するための言語である．

形式仕様は，形式仕様記述言語によって厳密に記述された仕様のことである．形式仕様は，非曖昧性，完全性，明晰性，反駁可能性などの性質が求められる．

形式仕様記述言語には，多種多様な数学モデル・数理体系に基づく言語が数多く提案されている．代表的なものとしては，公的記述法を用いた VDM[10]，代数的記述法を用いた,CafeOBJ,Maude 等があげられる．ここでは代数的手法に基づき仕様の対話的な演繹的検証が可能であり，また，探索的検証法の二つが可能である CafeOBJ について取り上げる．

2.2 代数仕様記述言語 CafeOBJ

代数仕様記述言語 CafeOBJ は，順序ソート項書換え論理に基づいた代数仕様記述言語である．等式を書換規則として解釈し，計算機上で実行することが可能であるため，対話的な検証法が可能である．

CafeOBJ の仕様は，代数モデルに基づく問題のモデル化である．仕様を構成する等式を書換規則として実行することができ，仕様の意味を規定する等式論理に忠実であるので，処理系でもある CafeOBJ によって対話的な検証をすることができる．順序ソート代数は汎用性のあるモデル化の枠組みであり，データ型や抽象機械などの現実のシステムを記述するのに有効なモデルを包含している．

CafeOBJ の特徴としては次のようなものがある．CafeOBJ 仕様はモジュール単位で記述され，協力なモジュール化機構を持つ．モジュールのパラメータ化やモジュールの輸入の際に名前の付け替えができ，中ぢょ独活の高いモデル化が可能である．ソート間に包含関係を作ることが出来，継承などの概念を扱うことが出来る．また，書換規則によって動的なシステムの記述が可能である．

CafeOBJ では，モジュールを一つの単位として仕様を記述する．モジュールはモジュール名と輸入，シグネチャ，公理から攻勢される，輸入はライブラリや既に記述されているモジュールを再利用する際に用いる．シグネチャは型宣言などで，ソートとソート上での演算で構成される．公理は等式やルール宣言などを記述する．次セクションでは，CafeOBJ の構文について説明する．

2.2.1 CafeOBJ 構文

CafeOBJ では，代数仕様と呼ばれる仕様を記述する．代数仕様はソートとソート上での演算で構成されるシグネチャにより言語を定義する．また，二項間の等価関係は，等式と変数から構成される公理によって定義する．CafeOBJ では，シグネチャと公理から構成されるモジュールを一つの単位として仕様を記述する．本節では，CafeOBJ の構文について説明する．

モジュール宣言

CafeOBJ で仕様を記述するには，まずモジュールを宣言し，その中に様々な宣言をする．モジュールは”mod”を使って次のように宣言する．

```
mod module_name {
  module\_element*
}
```

module_name にモジュール名を記述し，module_element にモジュールの構成要素を記述する．モジュールの構成要素とは，以降に説明する，シグネチャや公理のことである．

また，モジュールを宣言する際，モジュールの意味論を指定することが可能である．きつい意味論に基づいて記述するときには”mod!”を記述し，ゆるい意味論のときには”mod*”と記述する．特に意味論を指定しない場合には”mod”を用いる．

輸入宣言

モジュールの輸入宣言には”pr”，”ex”，”us”を用いる．輸入（import）とは，あるモジュール内で，別の定義済みモジュールの宣言を使用可能にすることをいう．それぞれ以下のように宣言する．

```
pr(module-exp)
ex(module-exp)
us(module-exp)
```

module-exp には輸入するモジュール名を記述する．”pr”，”ex”，”us”のそれぞれの輸入の宣言には異なる意味がある．意味の違いは，輸入元でのモジュールにおける変更の許容の違いである．”pr”はソートに対し新しい要素を付け加えることが出来ず，異なっていた二つの要素をひとしいものとすることができない．”ex”はソートに対し，新しい要素を付け加えることが出来るが，”pr”同様に異なっていた二つの要素を等しいものとすることができない．”us”は特に制限がない場合に用いる．

ソート宣言

代数においてソートとは，プログラミング言語の型に対応する概念であり，CafeOBJ では強く型付けされた言語である．CafeOBJ においてソートは”[...]”及び”*[...]*”を用いて宣言する．

```
[sort-name...sort-name]
*[sort-name...sort-name]*
```

sort-name にはソート名を記述する．ソート名をスペースで区切ることで，複数のソートを一度に宣言することができる．CafeOBJ には，二種類のソートが存在する．一つは可視ソートで，もう一つは隠蔽ソートである．可視ソートの表現は”[...]”で表現し，隠蔽ソートは”*[...]*”で表現する．また，複数のソートを宣言した場合，ソート間の包含関係はで示すことが出来る．可視ソートは始代数に基づく抽象データ型を，隠蔽ソートは隠蔽代数に基づく抽象機械の状態空間を表すのに適している．

オペレータ宣言

オペレータの宣言は以下のように”op”を用いて宣言する．

```
op Operation_Name : arity -> coarity
op Operation_Name : -> coarity
ops Operation_Name ... Operation_Name :arity -> coarity
bop Operation_Name : arity -> coarity
```

Operation_Name にはオペレータ名を記述する．arity,coarity はそれぞれ引数のソート及びその組，オペレータのもつソート（返り値）に対応している概念である．arity が無い場合，そのオペレータは定数をあらわす表現となる．arity と coarity の組をランクと言い，ランクが同じ場合，”ops”を用いて一度に宣言が出来る．また，隠蔽ソートを arity に持つ場合は,”op”の代わりに”bop”で宣言しなければならない．

等式宣言

等式の宣言は”eq”を用いて行う．また，条件付等式は”ceq”を用いて宣言を行う．

```
eq lhs = rhs .
ceq lhs = rhs if condition .
```

等式の宣言は左辺 (lhs) の項と，右辺 (rhs) の項の等価関係を宣言する．等式の最後にはピリオドを打たなければならない．また，条件付等式の場合，”if”以降にその等価関係が成り立つための条件を記述する．”condition”は内臓モジュール BOOL で宣言されている Bool ソート上の項でなければならない．これら仕様上に記述された等式は，公理として推論の際に使用される．

変数宣言

変数の宣言は”var”を用いて行う．

```
var var-name : sort-name .
vars var-name ... var-name : sort-name .
```

var-name には変数名を記述する．sort-name には変数のソートを記述する．また，複数同時に宣言したい場合は”vars”を用いて宣言できる．

遷移宣言

遷移の宣言は，”trans”で行う．しかし，通常の CafeOBJ による代数仕様の仕様記述では用いない．それは，”eq”を用いた書換規則で動的な仕様が記述可能だからである．だが，後に述べる Search コマンドによる検証を行う際には，必要となる．従って，後の節で解説する．

2.2.2 CafeOBJ/OTS 法

観測遷移システム（Obsevational Transition System）[11] は観測値の集まりで状態を表した状態遷移機械であり，観測等価が振舞等価と一致する振舞仕様のモデルである．代数仕様記述言語 CafeOBJ で OTS の定義に基づいてモデル化を行うことを CafeOBJ/OTS 法という．

OTS ではモデル化するシステムを隠蔽代数を用いてブラックボックスとして捉えることで，外部から観測可能な観測値に基づき，状態の性質を見ることが出来る．隠蔽代数により内部の状態記述を記述しないことから，仕様の記述が容易であるという利点がある．また，内部の状態を詳細に記述しない，または内部の実装による依存がないことから，抽象度の高いモデル化を可能にしている．本研究では CafeOBJ/OTS 法に基づいて記述された仕様に関して，演繹的検証及び探索的検証を行う．

以下に OTS の定義を示す．

OTS は，観測の集合 $\mathcal{O}$，初期状態の集合 $\mathcal{I}$，遷移規則の集合 $\mathcal{T}$ の組によって構成される．

- $\mathcal{O}$: 観測の集合

  各観測は，状態を引数に取り，その状態を構成する任意のデータ型を返す関数である．ただし各観測の値域がそれぞれ異なっても良い．

- $\mathcal{I}$: 初期状態の集合

  初期状態 $\mathcal{I}$ は，各観測に関する初期値を定める．

- $\mathcal{T}$:遷移規則の集合

  各遷移関数は，ある状態を引数に取り，遷移後の状態を返す関数である．遷移関数は効果と効力条件（effective condition）の組によって定義される．効果は，各観測の変化のことを指し，効力条件は，遷移が成立する条件を記述したブール型を返す事前条件のことである．

以上のような定義から，状態 u1 と状態 u2 が観測等価とは任意の観測関数 o に対して，o(u1) = o(u2) で定義される．OTS は，観測等価な状態 u1,u2, 遷移関数 t に対して状態 t(u1) = t(u2) が観測等価という性質を持つ．OTS 仕様では，遷移後の観測値がすべての遷移関数と観測関数の組み合わせについて定義されているかどうかが，十分完全性に対応する性質となる．

2.3 演繹的検証法

演繹的な検証法とは，対話的な検証法のことで，無限状態を持つ問題に対しても可能であるなど，非常に強力で広範な問題を取り扱うことが出来る．演繹的検証法では，人の手によって記述あれた証明を追って検証を行う．証明中に仕様や証明すべき性質に関し

て誤りがある場合，システムがユーザに対しエラーを報告し修正を促す．そして，修正した仕様や性質に対して検証を再度行う．そうやって，システムとユーザの対話的な検証によって検証を完成させる．演繹的な検証は，公理と推論規則から演繹的に証明していく．演繹的な検証は，証明結果の無矛盾性が保証されており，証明結果より新たな定理を追加することで検証能力を高めることができる．CafeOBJ は演繹的検証に相当する機能を持つ．それは証明譜による検証である．証明譜による検証を以下に示す．

証明譜による検証

CafeOBJ は証明譜による検証が可能である．証明譜とは推論と演繹によって証明手順が記述されたドキュメントである．証明譜は，証明したい性質に対し，人間の手によって場合分けや補題を導入し，システムがそれらに対して結果を返すことで，システムと対話的に証明していく手法をとる．検証は，等式による書換に基づき，Reduction コマンドをもちいて，性質の式を書き換えていき，最終的にシステムが真を返せば証明されたとされる．以下に例を示す．

```
--> ======================================================
--> send1(drop2(S)) = drop2(send1(S))
--> ======================================================
-- case
--   /empty?(cell1(s))
--   /~empty?(cell1(s))
-- ------------------------------------------------------
open SCPobEq
op s : -> Sys .
eq empty?(cell1(s)) = true .
-- |=
red send2(drop1(s)) =ob= drop1(send2(s)) .
close

open SCPobEq
op s : -> Sys .
eq empty?(cell1(s)) = false .
-- |=
red send2(drop1(s)) =ob= drop1(send2(s)) .
close
```

性質 send2(drop1(s)) =ob= drop1(send2(s)) を証明譜により検証していることを示す．この場合，場合分けは二つあり，empty?(cell1(s)) と not(empty?(cell1(s))) である．それ

ぞれ reduction を行う前に，等式により宣言を行い検証している．両方の場合ともシステムは，true を返すので，この性質は成り立ってることが証明されている．場合分けは，red send2(drop1(s)) =ob= drop1(send2(s)) を行い，結果として出される項から分析を行うことで得られる．場合分けを行ったときに，どちらかが ture を返す場合分けであれば，場合分けとしては妥当といえる．本研究では，この証明譜による検証を演繹的検証として用いる．

2.4 探索的検証法

近年，ソフトウェアの分野において，全探索による検証技術であるモデル検査が大きな成功を収めて注目されている．モデル検査は，与えられた仕様や性質に対し，全状態を網羅的に探索し検査することができる技術の総称である．ここでいう探索的検証法とは，モデル検査による検証を指す．

モデル検査による検証は，問題の記述能力・検証能力に制限があり，あらゆる問題に適用することができない．探索的検証法は，演繹的検証法に比べ，検証の自動化が可能である点，性質が成り立たない場合には反例を示すことが出来る点が強みであるといえる．しかし，モデル検査は全探索を行うため，無限状態をとるシステムを検証できない．また，現実の問題は非常に規模が大きく，状態爆破を起こし，時間的な問題などによって扱うことの出来ない問題もおおい．

モデル検査を扱えるシステムは多くあり，主に産業界で成功している SPIN が最も有名であると考えられる．また，CafeOBJ や Maude もモデル検査の機能を持つ．しかし，数あるモデル検査のなかでも，演繹的検証と探索的検証を同時に扱える機能を持つものは，現在 CafeOBJ しかない．CafeOBJ には，モデル検査をおこなうための Search コマンドという機能を持っている．本研究では CafeOBJ によるモデル検査を対象としているため，Search コマンドを次に説明する．

Search コマンドによるモデル検査

CafeOBJ には Search コマンドという機能をがある．Search コマンドを使用するためには，CafeOBJ のオペレータである trans が必要である．trans は遷移規則を表すオペレータで，次のように宣言する．

遷移宣言

モデル検査用の遷移の宣言は”trans”を用いて行う．また，条件付遷移は”ctrans”を用いて宣言を行う．

```
trans [trans-name] : < rhs > => < lhs > .
```

```
ctrans [trans-name] : < rhs > => < lhs > if condition .
```

左辺に遷移前の状態を記述し，右辺には遷移後の状態を記述する．trans-name には遷移規則の名前を記述する．また，条件付き遷移の condition の部分には，遷移の条件が成り立つ Bool ソート上の項を記述する．Search コマンドでは，遷移宣言を，左辺から右辺への遷移規則とし，遷移を行っていく．

Search コマンド

CafeOBJ の ver1.4.8 で新たに導入された組み込みの Search コマンドについて説明する．Search コマンドは書換論理における可能な遷移を網羅的に探索する機能を提供するものであり，組み込みモジュール RWL で定義されている．

Search コマンドは，red コマンドを用いて，項があるパターンへと一致する項へ遷移規則（trans）によって到達するか否かを調べることが出来る．組み込みモジュール RWL は，従来通り，ユーザの定義したモジュールが遷移規則の宣言（trans）を持っているときに自動的にインポートされる．従って，何かを定義しなければならないということは必要ない．以下に Search コマンドの構文と挙動を説明する．

構文

```
t1 =(m,n)=>* t2 suchThat pred(t2)
```

結果のソートは BOOL 型である．項 t1 が trans による書換によって，t2 で指定される項にパターンに一致するような項へ遷移する場合に true を，そうでない場合に false を返す．suchThat がある場合は，suchThat 以降の pred(t2) が true の場合にパターンに一致したと見なす．項 t1，項 t2 及び suchThat 以降の pred(t2) は任意の項で良い．ただし，suchThat に含まれる変数は t2 に含まれるサブセットでなければならない．

m は結果上限を示し，n は探索する深さを示す．m 及び n は 0 より大きい自然数かもしくは*でなければならない．*は上限なしを意味する．また，*のほかに+と!が存在する．

動作

- t1 =(m,n)=>* t2

  t1 =(n,m)=>* t2 は項 t1 が項 t2 で示されるパターンに 0 回以上の遷移で到達する場合に true を返し，そうで無い場合は false を返す．遷移は幅優先探索によって求められる，すんわち，t1 に対して可能な全ての遷移を求めるという展開を行う．この深さに対する上限を指定するのが n である．また m は発見された解の数の上限である．上限の制限をかけた場合，どちらかが上限に達し次第探索は終了する．その間に一つでも解が得られれば true を，そうでない場合には false を返す．

- t1 =(m,n)=>* t2 suchThat pred(t2)

  t1 =(m,n)=>* t2 suchThat pred(t2) を実行した場合，基本的には上記の動作と同じであるが，t1 が遷移して t2 にパターンに一致した際に，そこで得られた t2 の項を t3 と比較し，それらを簡約した結果が true であれば実際にパターンに一致したとみなし，そうで無い場合は一致していないとみなす．

- t1 =(m,n)=>+ t2

  *の代わりに，+を用いた場合，項 t1 が項 t2 に一回以上の遷移で到達する場合に false を返す．それ以外の動作は，*と同じである．また suchThat を末尾に追加したならば，上記の条件一致の動作と同じになる．

- t1 =(m,n)=>! t2

  !を用いた場合，t2 は項 t1 がそれ以上の遷移が無い項に到達した際に，t2 で指定されたパターンに一致する場合に true を返し，そうで無い場合 false を返す．それ以外の動作は，*と同じである．また suchThat を末尾に追加したならば，上記の条件一致の動作と同じになる．

以下に Search コマンドの動作のまとめを示す．

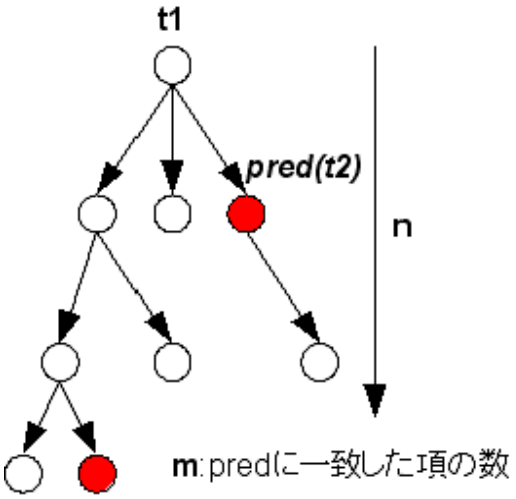

図 2.1: Search コマンドの動作例

t1 が遷移の初期状態で，白丸が任意の遷移後の項を示し，黒丸（赤丸）が t2 及び suchThat pred(t2) を適用し，相当すると判断された項を状態を示す．各状態に対して，各遷移を適用していくので，Search コマンドによる探索数は，階乗数ずつ増えていくことが分かる．

モデル検査としての Search コマンド

Search コマンドとモデル検査は厳密には同じではない．しかし，モデル検査による検証の手法及び道具として aerch コマンドを活用することは出来る．モデル検査は網羅的に探索をし，証明したい性質の反例や間違いなどを発見する，また Search コマンドも記述の仕方によって網羅的に全探索するし，反例を発見することが出来る．

モデル検査として全探索を目的とした場合，Search コマンドは，t1 =(m,n)=>* t2 suchThat pred(t2) を用いることが一般的である．全探索する場合，m 及び n には上限なしの*を用い，suchThat 以降の述語記述には，証明したい性質を記述する．その際，suchThat (not pred(t2)) と記述することで，証明したい性質が成り立たない状態を検証する，つまり反例を検証することが出来る．もし true が出た場合，反例が発見されたということを示し，反例の具体的な状態が検出される．また，false が出た場合，それは反例が無いということを示す．*を用いているので，全ての状態（到達可能状態）に対し遷移を行った結果であることが分かる．また，それは到達可能状態が無限である場合には，全探索をした場合，有限時間で解が出ないことを示す．ただし，深さを指定すれば，到達可能状態が無限であっても，そこまでの解を求めることは可能である．

第3章　演繹的な検証法と探索的な検証法の組み合わせ

本研究では，演繹的な検証法と探索的な検証法を組み合わせて，従来の検証法より効率的な検証法を提案する．組み合わせた検証法とは，モデル検査が扱えない無限状態の仕様に対し，有限化を行い検証可能にすることである．有限化の手法として，等式による状態の有限化を行い，それら等式の正当性は証明譜で行う手法である．両検証法の特徴やメリットデメリットをうまく用いて検証の効率を上げた方法である．本章では，相互排除プロトコル QLOCK に対し，本検証を用いた場合に関して説明する．

3.1　相互排除プロトコル QLOCK

本節では相互排除プロトコルを例に，検証法の説明を行う．よって，まず相互排除プロトコルである QLOCK に関して説明を行う．

QLOCK

複数のプロセスまたはエージェントにより共有される資源がある場合，任意の時点においてその資源を使用しているプロセスは高々一つである，という意味において，資源を排他的に使用することを要求されることが多い．その資源を排他的に共有する仕組みが相互排除プロトコルという．

QLOCK とは，この相互排除プロトコルをアトミックな待ち行列を用いて実現したプロトコルである．アトミックな待ち行列とは，待ち行列における要素の操作（追加，削除）などが不可分であることを示す．

QLOCK の各プロセスは以下の擬似コードで記述されるように振舞う．

```
Loop {
        Remainder Section;
    rm: put(queue,i);
    wt: repeat until top(queue) = i;
        Critical Section;
    cs: get(queue)
```

```
}
```

queue は，すべてのプロセスによる共有されるアトミックな待ち行列である．put,top 及び get は，待ち行列の操作であり，その操作の途中に他のいかなる操作も割り込むことがないという意味で，不可分に実行される．

各プロセス i は，共有資源を使用しない場合は，その他の領域（Remainder Section）におり，使用するときわどい領域（Critical Section）にいる．各プロセス i は，共有資源を使用したいとき，まず，プロセスの識別子 i を queue の末尾に追加する．put(queue,i) はそれを現している．そして，queue の先頭にプロセス i が来るまで待ち，きわどい領域に入る．repeat until top(queue,i) = i がきわどい領域に入るまでの繰り返しである．共有資源を使用した後は，queue の先頭を取り除くことを示している get(queue) を実行し，プロセス i は最初の状態であるその他の領域に戻る．各プロセス i は以上のとこを繰り返す．

rm,wt 及び cs はラベル (Label) である．プロセスがその他の領域にいる状態とき，そのプロセスのラベルは rm になり，queue に入りきわどい領域を待っている状態のときのラベルは wt になり，プロセスがきわどい領域にいるときのラベルは cs となる．

初期状態では，全てのプロセスのラベルは rm になり，また queue は空の状態である．

相互排除プロトコルを例にとり，検証法の説明を行っていく．

3.1.1 モデリング

相互排除性プトロコル QLOCK を，CafeaOBJ/OTS 法に基づきモデル化を行っていく．OTS とは，初期状態と観測の集合，遷移規則の集合の組によって定義される状態遷移機械である．

観測の定義

QLOCK の状態を特徴付ける観測値は，待ち行列（Queue）の状態と，各プロセスが持つラベルである．S_{QLOCK} の状態を引数とし，これらの値を返す，pc, queue を定義する．pc は,S_{QLOCK} の状態と任意のプロセス i を引数に取り，プロセス i における rm,wt,cs のいずれかのラベルを返す．queue は，S_{QLOCK} の状態を引数に取り，Queue の状態を返す．

よって，S_{QLOCK} の観測状態の集合 $\mathcal{O}_{QLOCK}$ は次のような定義になる．

$$\mathcal{O}_{QLOCK} \triangleq \{\ \text{queue} : \text{Sys} \rightarrow \text{Queue},\ \text{pc} : \text{Sys Pid} \rightarrow \text{Label}\ \}$$

遷移の定義

遷移の定義は擬似コードから，次の三つと考えられる．

(1)put(queue,i) の実行，(2)repeat until top(queue) = i の一回分の繰り返し，(3)get(queue) の実行である．これらをそれぞれ遷移関数 want,try, および exit で表す．それぞれの遷移

関数は，任意のプロセス i と S_{QLOCK} の状態を引数に取り，それぞれの関数を実行した後の状態を返す．want は (1) にあたり，任意のプロセス i を Queue の末尾に追加する遷移である．try は (2) にあたり，任意のプロセスが先頭に来るまでリピートする遷移である．exit は (3) にあたり，任意のプロセスがきわどい領域にいれば，Queue から削除するという遷移である．

よって，S_{QLOCK} の観測状態の集合 $\mathcal{T}_{QLOCK}$ は次のような定義になる．

$$\mathcal{T}_{QLOCK} \triangleq \{\ \text{try} : \text{Sys Pid} \to \text{Sys}, \text{want} : \text{Sys Pid} \to \text{Sys}, \\ \text{exit} : \text{Sys Pid} \to \text{Sys}\ \}$$

初期状態の定義

S_{QLOCK} の状態 init では，queue(init) では，空の待ち行列を返し，任意のプロセス i に対し，pc(init,i) ではラベル rm を返す．したがって，S_{QLOCK} の初期状態の集合 $\mathcal{I}_{QLOCK}$ は次のように定義できる．

$$\mathcal{I}_{QLOCK} \triangleq \{\ \text{init} \mid \text{queue(init)} = \text{empty} \wedge \forall_i : \text{Pid.(pc(init,i))} = \text{rm}\ \}$$

CafeOBJ で記述するにおいて OTS に基づく QLOCK の振舞の定義（遷移の定義）には，効力条件の定義が必要である．効力条件とは，振舞の事前条件の定義のことを指す．効力条件を示し，遷移を定義することによって，QLOCK の OTS 仕様は完成する．また，観測の定義は，上記で説明した通りに記述すればよいが，初期状態の定義は等式を用いて記述する必要がある．よって，CafeOBJ/OTS 法上における初期状態の定義と，効力条件を伴った遷移演算の定義を以下に説明する．

初期状態の定義

任意の初期状態を表す定数 init は，以下の通りに定義される．これらの等式は $\mathcal{I}_{QLOCK}$ の定義を CafeOBJ で記述したものに相当する．

```
eq pc(init,I) = rm .
eq queue(init) = empty .
```

遷移演算 want の定義

遷移 want を定義する等式及び効力条件は次の通りの定義される．

```
op c-want : Sys Pid -> Bool
eq c-want(S,I) = (pc(S,I) = rm) .
--
ceq pc(want(S,I),J) = (if I = J then wt else pc(S,J) fi) if c-want(S,I) .
```

```
ceq queue(want(S,I)) = put(I,queue(S)) if c-want(S,I) .
ceq want(S,I) = S if not c-want(S,I) .
```

演算 c-want は，遷移演算 want の効力条件を示している．効力条件は最初の等式で定義されている．続く二つの等式で，条件が満たされた場合，何が起きるかを各観測関数ごとに示している．

効力条件が満たされれば，事後状態 want(S,I) に対して，pc はプロセス I に対して wt を返し，I 以外のプロセスに対し状態 S と同じ値 pc(S,J) を返す．queue は状態 S における待ち行列 queue(S) にプロセス I を末尾に加えた待ち行列を返す．最後の等式には，効力条件が満たされない場合，何も起きないことを定義している．

遷移演算 try の定義

遷移 try を定義する等式及び効力条件は次の通りの定義される．

```
op c-try : Sys Pid -> Bool
eq c-try(S,I) = (pc(S,I) = wt and top(queue(S)) = I) .
--
ceq pc(try(S,I),J) = (if I = J then cs else pc(S,J) fi) if c-try(S,I) .
eq  queue(try(S,I)) = queue(S) .
ceq try(S,I) = S if not c-try(S,I) .
```

遷移演算 exit の定義

遷移 exit を定義する等式及び効力条件は次の通りの定義される．

```
op c-exit : Sys Pid -> Bool
eq c-exit(S,I) = (pc(S,I) = cs) .
--
ceq pc(exit(S,I),J) = (if I = J then rm else pc(S,J) fi) if c-exit(S,I) .
ceq queue(exit(S,I)) = get(queue(S)) if c-exit(S,I) .
ceq exit(S,I) = S if not c-exit(S,I) .
```

相互排除性

観測遷移システム S_{QLOCK} の到達可能状態とは，初期状態から有限個の状態遷移により到達できる状態である．S_{QLOCK} の全ての到達可能状態で成り立つ状態述語を S_{QLOCK} の不変性と呼ぶ．QLOCK には満たすべき性質がある．それは，きわどい領域にいるプロセスは常に高々一つである，ということを意味する相互排除性である．CafeOBj 上で相互排除性をモデル化すると以下のような式になる．

```
eq mutualEx(S,I,J) = (pc(S,I) = cs and pc(S,J) = cs) implies I = J .
```

状態 S とある二つのプロセスを引数にとり，プロセス I がきわどい領域におり，またプロセス J もきわどい領域にいるならば，プロセス I とプロセス J は同じである，つまり，きわどい領域にいるプロセスは高々一つであることを示している．

プロセスの有限化

Search コマンドを用いたモデル検査による検証（全探索の検証）を行うためには，到達可能状態の有限化を行わなければならない．到達可能状態が有限であれば，状態空間自体は無限でも全探索のモデル検査が可能になる．したがって，QLOCK において無限状態をとると考えられる要素は，プロセス数の数であると考えられる．そこでプロセスの数を有限化（1,2,3,4）することを考える．

```
mod* PID {
  [PidConst < Pid < PidErr]
  op none : -> PidErr
  pr(EQL)
  vars Ic1 Ic2 : PidConst
  eq (Ic1 = Ic2) = (Ic1 == Ic2) .
  eq (none = I:Pid) = false .
}
```

ソートに PidConst を追加し，Pid の数を定数化する．また，等号にはビルドインモジュール EQL を輸入し，それらを用いて定義するようにする．有限化されたプロセス数を用いるためには，仕様において定数で必要な数の分だけ宣言する必要がある．

```
mod* PID {
ops i j k l m : -> PidConst .
```

以上の式を記述すればプロセス数は 5 つに有限化される．また，これらの式を追加すれば到達可能状態は有限になるが，状態空間自体は無限になる．それは，待ち行列の長さが無限になっているからである．

3.2 モデル検査への適用

CafeOBJ/OTS 法で記述された仕様を検証する場合には，trans を記述し，Search コマンドを用いて検証する必要がある．遷移規則 (trans) を記述することにより，OTS のダイレクトなモデル検査が可能になる．

モデル検査用の仕様への変換

以下に遷移規則 (trans) の追加の方法を示す.

```
mod! QLOCKijklTrans {
  pr(QLOCK)
  ops i j k l : -> PidConst .

  [ Config ]
  op <_> : Sys -> Config .
  var S : Sys .

  ctrans [want-i] : < S > => < want(S,i) > if c-want(S,i) .
  ctrans [want-j] : < S > => < want(S,j) > if c-want(S,j) .
  ctrans [want-k] : < S > => < want(S,k) > if c-want(S,k) .
  ctrans [want-y] : < S > => < want(S,l) > if c-want(S,l) .

  ctrans [try-i] : < S > => < try(S,i) > if c-try(S,i) .
  ctrans [try-j] : < S > => < try(S,j) > if c-try(S,j) .
  ctrans [try-k] : < S > => < try(S,k) > if c-try(S,k) .
  ctrans [try-y] : < S > => < try(S,l) > if c-try(S,l) .

  ctrans [exit-i] : < S > => < exit(S,i) > if c-exit(S,i) .
  ctrans [exit-j] : < S > => < exit(S,j) > if c-exit(S,j) .
  ctrans [exit-k] : < S > => < exit(S,k) > if c-exit(S,k) .
  ctrans [exit-y] : < S > => < exit(S,l) > if c-exit(S,l) .
}
```

以上の式は，QLOCK のプロセス数が 4 つのモデル検査用仕様である．pr(QLOCK) でもともとの CafeOBJ/OTS 法による QLOCK の仕様を読み込む．オペレータ宣言において i,j,k,l は前述したプロセス数の有限化（4 個）を示している．

また，ソート宣言で Configuration というデータ構造を宣言する．ここでの Configuration は状態 S である．それぞれ 4 つのプロセスに対し遷移規則（Trans）を宣言する．want(S,I) は引数にプロセスをとるため，プロセス数の個数分のパターンが必要になるので，ここでは,i,j,k,l の 4 つ分の遷移宣言を行う．また，同様に try,exit も宣言を行う．プロセス数 5 つ，6 つの場合も同様に記述することで，モデル検査用の仕様に変換することができる．if 以降の条件は効力条件を記述する．以上のような容易な OTS の容易な変換によって，モデル検査による検証を行うことができる．

Search コマンドの実行の準備

モデル検査用の仕様に変換した後，Search コマンドを用いて検証を行う．Search コマンドを用いてモデル検査を行うためには，以下のように行う．

モデル検査用の仕様

```
open (QLOCKijklTrans + MEX)
pred mutualEx-ij : Sys .
eq mutualEx-ij(S:Sys) = mutualEx(S,i,j) .
red < init > =(*,1)=>* < S:Sys > suchThat (not mutualEx-ij(S)) .
red < init > =(*,2)=>* < S:Sys > suchThat (not mutualEx-ij(S)) .
red < init > =(*,3)=>* < S:Sys > suchThat (not mutualEx-ij(S)) .
red < init > =(*,4)=>* < S:Sys > suchThat (not mutualEx-ij(S)) .
red < init > =(*,5)=>* < S:Sys > suchThat (not mutualEx-ij(S)) .
close
```

QLOCK のプロセス数が 4 つの仕様を QLOCKijklTrans とし，相互排除を定義した仕様を MEX とする．モデル検査を行うためにそれぞれ読み込む．また，相互排除性の引数を状態 S だけに簡潔化するために mutualEx-ij の定義を行う．red からはじまる部分がモデル検査で，それぞれ全探索ではなく，深さを指定して検査を行っている．

MEX

また，MEX としている相互排除性のモジュールは以下のようになる．

```
mod MEX {
 pr(QLOCK)
 pred mutualEx : Sys Pid Pid .
 var S :  Sys . vars I J : Pid .
 eq mutualEx(S,I,J) = ((pc(S,I) = cs and pc(S,J) = cs) implies I = J) .
}
```

モデル検査の実行

Search コマンドを用いたモデル検査の実行の様子を以下に示す．

```
_*
-- opening module QLOCKobEq + QLOCKijklTrans + MEX.. done._*
-- reduce in %QLOCKijklTrans + MEX : ((< init >) = ( * , 1 )
```

```
                    =>* (< S >) suchThat (not mutualEx-ij(S))):Bool
-- reached to the specified search depth 1.
(false):Bool
(0.000 sec for parse, 669 rewrites(0.000 sec),
                         984 matches, 161 memo hits)
-- reduce in %QLOCKijklTrans + MEX : ((< init >) = ( * , 2 )
                    =>* (< S >) suchThat (not mutualEx-ij(S))):Bool
-- reached to the specified search depth 2.
(false):Bool
(0.000 sec for parse, 2524 rewrites(0.032 sec),
                          3699 matches, 670 memo hits)
-- reduce in %QLOCKijklTrans + MEX : ((< init >) = ( * , 3 )
                    =>* (< S >) suchThat (not mutualEx-ij(S))):Bool
-- reached to the specified search depth 3.
(false):Bool
(0.000 sec for parse, 8480 rewrites(0.140 sec),
                          12363 matches, 2302 memo hits)
-- reduce in %QLOCKijklTrans + MEX : ((< init >) = ( * , 4 )
                    =>* (< S >) suchThat (not mutualEx-ij(S))):Bool
-- reached to the specified search depth 4.
(false):Bool
(0.000 sec for parse, 23368 rewrites(0.485 sec),
                           33963 matches, 6498 memo hits)
-- reduce in %QLOCKijklTrans + MEX : ((< init >) = ( * , 5 )
                    =>* (< S >) suchThat (not mutualEx-ij(S))):Bool
-- reached to the specified search depth 5.
(false):Bool
(0.000 sec for parse, 58432 rewrites(1.984 sec),
                           84939 matches, 16522 memo hits)
_*
```

以上は，プロセス数 4 つの仕様に関して実行を行った結果である．深さを一つずつ増やして探索している．全て False の結果を返している．それは，反例が発見されなかった，つまり相互排除性を満たさない状態は深さ n までは無かったということを示している．

False の後に出ている記述は，書換回数と結果が出るまでにかかった時間が記述されている．プロセス数 3 では深さ 11 で，263 秒かかって結果が出る．また，プロセス数 4 では深さ 8 で 256 秒かかり結果が出る．さらに，プロセス数 5 つでは深さ 6 で 136 秒で結果が出る．ぞれそれのプロセスでそれら以上の深さを検査しようとすると，メモリ領域が足りなかったり，結果が出るまでに数十時間かかる等の問題が起こる．

全探索における問題

モデル検査による検証は網羅的に全探索を行うことで真価を発揮する．上記の QLOCK の仕様に対し，全探索を行わないと，反例が深さ x において出るかもしれない，という疑念を払拭することが出来ない．しかし，上記 QLOCK の仕様に対し，全探索（深さを指定しない探索）を行うと，結果が出てこないという問題が発生する．Search コマンドは到達可能状態において検査を行う．よって，到達可能状態を有限化したことで，全探索が可能になると考えられる．しかし，実行結果では結果が出てこない．それは，項に問題がある．

以下の実行結果は，解の個数を 201 まで出した結果である．

```
** Found [state 198] (< want(try(want(exit(try(
                               want(init,i),i),i),i),i),j) >):Config
   { S:Sys |-> want(try(want(exit(try(want(init,i),i),i),i),i),j) }

** Found [state 199] (< want(try(want(exit(try(
                               want(init,i),i),i),i),i),k) >):Config
   { S:Sys |-> want(try(want(exit(try(want(init,i),i),i),i),i),k) }

** Found [state 200] (< exit(try(want(exit(try(
                               want(init,i),i),i),i),i),i) >):Config
   { S:Sys |-> exit(try(want(exit(try(want(init,i),i),i),i),i),i) }
```

上の結果から，プロセス i が want,try,exit を繰り返していることが分かる．プロセス i が want,try,exit を二週行った後の観測値は，プロセス i が want,try,exit を行った後の観測値と同じであり，初期状態（init）の観測値と同じである．つまり，状態（観測値の集合）が同じであっても，それらを表現する項が違うと違う状態と見なされ，それらが，プロセス i が待ち行列に何回も入ったり出たりという様に無限に存在すると，解が出ないと考えられる．

全探索を行うためには，このような問題を解決する必要があり，解決する方法としては二つ考えられる．一つは withStateEq と観測等価を用いた方法であり，もう一つは等式による状態の削減を行う方法である．

3.3 withStateEq と観測等価

Search コマンドに関する条件で withStateEq と観測等価関数を用いることにより，同じ状態を表現する項の有限化を行うことができる．

withStateEq

以下に Search コマンドの witStateEq 条件を説明する．

```
t1 =(m,n)=>* t2 withStateEq pred(S1:Sort,S2:Sort)
```

withStateEq 条件とは，withStateEq 以降の述語 (pred) において，新しく検索した項 (S2) が，以前に検索した項 (S1) と等価であるということを示している．pred には BOOL 型ソートを返す，S1 と S2 を引数に持つ述語を記述する．S1 及び S2 のソートは項 t2 と同等のソートもしくは項 t2 の上位関係にあるソートでなければならない．

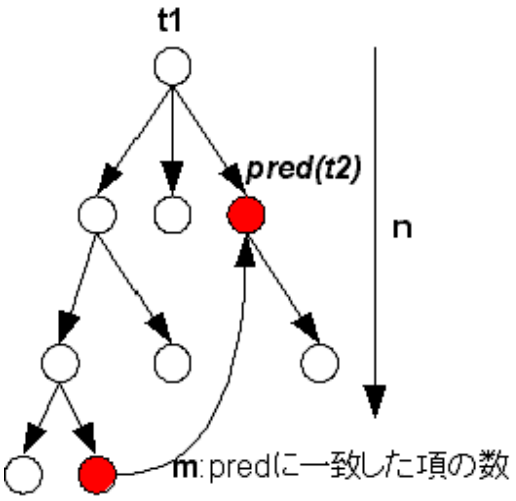

図 3.1: withStateEq の動作例

黒丸（赤丸）が withStateEq のある述語において一致した項であり，また，左側が新しく検索した項 S2 とし，右側のが，以前に検索した項 S1 とする．すると，S1 と S2 の状態を同じと見なし探索枝が統合され，次の探索に移る．

観測等価

withStateEq を用いて，ある述語において，新しく検索した項が以前に検索した項と同等であると振舞える．同じ状態を表現する項を有限化するためには，ある述語に対し，項が違えども同じ状態であることを示す述語を作る必要がある．ここでいう状態とは観測値の集合のことを示している．よって，新しく検索した項の観測値が，以前に検索した観測値と同じである，ということを示す関数が必要になることがわかる．それが，観測等価関数である．

観測等価関数とは，観測値の集合の全ての観測に対し，同じ値を返しているかどうかを判断する BOOL 型のソートを返す関数である．QLOCK の観測関数は pc と queue の二

つであり，これらが同じ値を返すかどうかを判断する関数を作ればよい．以下に QLOCK の観測等価関数を示す．

```
mod QLOCKobEq {
  pr(QLOCK + PIDlist)
  pred (_=_) : Queue Queue {comm}
  vars Q Q1 Q2 : Queue . vars X Y : Pid .
  eq (Q = Q) = true .
  eq ((Q1,X) = (Q2,Y)) = ((X = Y) and (Q1 = Q2)) .

  pred (_=ob=_with_) : Sys Sys PidList {memo} .
  pred (_=pc_with_) : Sys Sys PidList .

  vars S S1 S2 : Sys .
  var P : Pid .
  var PL : PidList .

  eq (S1 =pc S2 with P) = (pc(S1,P) = pc(S2,P)) .
  eq (S1 =pc S2 with (P PL)) =
             (pc(S1,P) = pc(S2,P)) and (S1 =pc S2 with PL) .
  eq (S1 =ob= S2 with PL) =
             (S1 =pc S2 with PL) and (queue(S1) = queue(S2)) .
}
```

=ob= with が観測等価関数になる．=ob=を定義するためには，=pc with という関数が必要になる．=pc with は任意の複数のプロセスにおいて，それぞれのプロセスのラベルが同じ値を返すかどうかを判断する関数である．=pc with を用いて=ob= with を定義すると，任意の複数のプロセスにおいて，それぞれのプロセスのラベルが同じ値を返し，かつ待ち行列の中身が同じである，という関数になる．with の部分にはプロセスのリストを引数に取る必要がある．例えば，i,j,k の三つで有限化したプロセスにおいては，S1 =ob= S2 with (i j k) と記述する．

withStateEq と観測等価関数を用いることにより，全ての観測値が同じである観測等価な項同士が統合され，全探索が出来るようになる．これらを用いたモデル検査による検証は以下のように記述する．

```
open (QLOCKobEq + QLOCKijkTrans + MEX)
pred mutualEx-ij : Sys .
eq mutualEx-ij(S:Sys) = mutualEx(S,i,j) .
pred _=ob=_ : Config Config {memo} .
vars S1 S2 : Sys .
```

```
eq (< S1 > =ob= < S2 >) = (S1 =ob= S2 with (i j k)) .
red < init > =(*,*)=>* < S:Sys > suchThat (not mutualEx-ij(S))
                                  withStateEq (C1:Config =ob= C2:Config) .
close
```

Configuration に一致させるため，Configuration を引数にとる観測等価を宣言する必要がある．ここでの Configuration は状態 S である．また，いくつか実行した結果を以下に示す．

```
-- opening module QLOCKobEq + QLOCKijkTrans + MEX.. done.__
-- reduce in %QLOCKobEq + QLOCKijkTrans + MEX :
             ((< init >) = ( * , * ) =>* (< S >)
                          suchThat (not mutualEx-ij(S))
                          withStateEq (C1 =ob= C2)):Bool
** No more possible transitions.
(false):Bool
(0.000 sec for parse, 20433 rewrites(0.313 sec),
                      32774 matches, 7673 memo hits)


-- opening module QLOCKobEq + QLOCKijklTrans + MEX.. done.__
-- reduce in %QLOCKobEq + QLOCKijklTrans + MEX :
             ((< init >) = ( * , * ) =>* (< S >)
                          suchThat (not mutualEx-ij(S))
                          withStateEq (C1 =ob= C2)):Bool
** No more possible transitions.
(false):Bool
(0.000 sec for parse, 397861 rewrites(8.953 sec),
                      634435 matches, 142169 memo hits)
```

以上の結果は，プロセス数 3 における全探索とプロセス数 4 の全探索である．それぞれ false を返していることから，反例が無いことを示している．プロセス数 3 つで探索にかかる時間は 0.3 秒であり，プロセス数 4 つで探索にかかる時間は約 9 秒である．プロセス数 5 つでは状態が爆発的に増え，メモリ領域や時間的な問題で解が返らないことがある．

3.4 等式による状態の削減

全探索を行うための方法として withStateEq と観測等価を用いた方法があるが，もう一方で等式によって状態を削減して探索する方法がある．それを以下に説明する．

同じ状態を表す項を有限化するために，等式を追加することによって，項の表現の有限化を目指す．項は遷移演算の重なりであることから，項の表現の有限化とは，遷移演算の適用のパターンを有限化することであると考えられる．つまり，ここで追加する等式は，ある遷移演算のパターンに対し，それとは異なる様々な遷移演算を適用しても同じ状態になることを示す等式である．

等式は eq を用いて宣言し，遷移演算の適用パターンのみ記述する．観測関数は用いない．その等式の意味は，左辺の遷移のパターンを適用しても右辺の遷移パターンを適用した状態と同じであり，左辺の遷移パターンは右辺の遷移パターンに常に書き換えられる．つまり，同じ状態を表現する項を有限化できると考えられる．QLOCK において，その等式は三つ考えられる．

- ceq try(want(S,I),J) = want(try(S,J),I) if not(I = J) .

  I と J が異なるとし，want(S,I) の後に try(S,J) を行った状態と try(S,J) の後に want(S,I) を行った状態が同じであることを示している．want(S,I) を行うと，プロセス I ラベルは wt になる，また try(S,J) を行ったあとのプロセス J のラベルは cs になる．try(S,J) が起きるということは，try が起きる前の状態 S でのプロセス J のラベルは wt である．また，同様に want(S,I) が起きる前のプロセス I のラベルは rm である．

  状態 S における待ち行列においては，プロセス J を先頭に中身が入っているか，プロセス J のみの状態が考えられる．どちらの場合も，want(S,I), try(S,J) の遷移に影響は無く，またプロセス J 含め中身の変動はない．

  従って，try(want(S,I),J) における各プロセスのラベル，want(try(S,J),I) における各プロセスのラベルを考えるだけでよく，どちらもプロセス I が wt で，プロセス J が cs になるため，この等式は成り立つと考えられる．

- ceq exit(want(S,I),J) = want(exit(S,J),I) if (not(I = J) and not(empty?(queue(S)))) .

  I と J が異なりかつ待ち行列の中身が空ではないとし，want(S,I) の後に exit(S,J) を行った状態と，exit(S,J) を行った後に want(S,I) を行った状態が同じであることを示している．want(S,I) を行うと，プロセス I ラベルは wt になる，また exit(S,J) を行ったあとのプロセス J のラベルは rm になる．exit(S,J) が起きるということは，try が起きる前の状態 S でのプロセス J のラベルは cs である．また，同様に want(S,I) が起きる前のプロセス I のラベルは rm である．

  待ち行列においては，プロセス J を先頭に中身が入っている状態のみである．この場合は，want(S,I), exit(S,J) の遷移に影響は無く，遷移後はプロセス J が待ち行列から削除され，中の要素が一つずつずれることになる．

  従って，exit(want(S,I),J) における各プロセスのラベル，want(exit(S,J),I) における各プロセスのラベルを考えるだけでよく，どちらもプロセス I が wt で，プロセス J が rm になるため，この等式は成り立つと考えられる．また，条件の not(empty?(queue(S)))

は自明であるように思えるが，後の証明の簡潔化に必要になる．もしこの条件がなかったとしてもモデル検査において問題はない．

- eq exit(try(want(init,I),I),I) = init .

  同一のプロセスが want,try,exit と遷移を行ったあとは，初期状態と同じであることを示している．この等式は比較的自明であり，want,try,exit 後の待ち行列は空で，またプロセスのラベルも rm である．よってこの等式も成り立つことが分かる．

以上の等式を追加すれば，状態が削減され，同じ状態を表現する項は有限化される．等式の発見や等式の分類に関しての分析は，通信プロトコルの章で述べる．

等式の証明

等式は思考実験やモデルの理解により発見されることが主であるが，思考しただけでは，必ずしもその等式が成り立つと判断できない．よって，等式の正当性を証明する必要がある．等式の正当性を証明する手段として証明譜による検証が考えられる．

withStateEq を用いたモデル検査で検証も可能だが，CafeOBJ/OTS 法で記述された仕様に対して，有限化したモデル検査と無限状態における証明譜の検証を行う仕様はほとんど変換を行う必要が無い．さらに，モデル検査においてはプロセスの数が変動する（3 個，4 個など）可能性があるため，任意のプロセスではなく無限の個数をとる仕様に対して証明できるのは強力である．また，withStateEq だと時間的な問題から解が出ない可能性がある．以上のことから証明譜を用いて検証するのだが，ここが演繹的な検証法と探索的な検証法の組み合わせた要素であるといえる．以下に証明譜による検証を示す．

等式の変換

まず，等式を証明譜で証明するためには，証明譜で検証できる形に変換しなければならない．等式のままだと，隠蔽代数同士の比較になるため，状態 S = 状態 S という形になっていまい，証明が出来ない．

よって，それぞれの状態における観測値を比較するようにしなければならない．つまり，観測等価関数を用いる必要がある．観測等価関数は 3.3 で説明を行っている．等号記号を=ob= with の形に置き換える．また，if 条件以降の式については，implication を用いて式の先頭に持ってくる必要がある．もし〜ならば，という意味は，implication と論理的に同等の意味である．それぞれ変換した結果を以下に示す．

- not(i = j) implies (try(want(s,i),j) =ob= want(try(s,j),i) with (i j))
- (not(i = j) and not(empty?(queue(s)))) implies (exit(want(s,i),j) =ob= want(exit(s,j),i) with (i j))

- exit(try(want(init,i),i),i) =ob= init with (i)

証明譜

それぞれ変換した等式を証明譜によって正当性を証明する．その証明譜を以下に示す.

```
open QLOCKobEq
op s : -> Sys .
ops i j : -> Pid .
eq i = j .
-- |=
red not(i = j) implies (try(want(s,i),j) =ob= want(try(s,j),i) with (i j) ) .
close
--> 1,~i=j,c-want(s,i),c-try(s,j)
open QLOCKobEq
op s : -> Sys .
ops i j : -> Pid .
--
eq (i = j) = false .
-- eq c-want(s,i) = true .
eq pc(s,i) = rm .
-- eq c-try(s,j) = true .
eq pc(s,j) = wt .
op q : -> Queue .
eq queue(s) = q,j .
-- |=
red not(i = j) implies
    (try(want(s,i),j) =ob= want(try(s,j),i) with (i j) ) .
close
--> 1,~i=j,c-want(s,i),~c-try(s,j),pc(s,j)=wt,queue(s)=empty
open QLOCKobEq
op s : -> Sys .
ops i j : -> Pid .
--
eq (i = j) = false .
-- eq c-want(s,i) = true .
eq pc(s,i) = rm .
--
eq c-try(s,j) = false .
```

```
-- eq ((pc(s,j) = wt) and (top(queue(s)) = j)) = false .
--
eq pc(s,j) = wt .
--
eq queue(s) = empty .
-- |=
red (not(i = j) implies
    (try(want(s,i),j) =ob= want(try(s,j),i) with (i j))) .
close
--> 1,~i=j,c-want(s,i),~c-try(s,j),pc(s,j)=wt,queue(s)=q,k
open QLOCKobEq
op s : -> Sys .
ops i j : -> Pid .
--
eq (i = j) = false .
-- eq c-want(s,i) = true .
eq pc(s,i) = rm .
--
eq c-try(s,j) = false .
-- eq ((pc(s,j) = wt) and (top(queue(s)) = j)) = false .
--
eq pc(s,j) = wt .
--
op q : -> Queue .
op k : -> Pid .
eq queue(s) = q,k .
eq (j = k) = false .
-- because  eq c-try(s,j) = false .
-- that is: eq ((pc(s,j) = wt) and (top(queue(s)) = j)) = false .
-- |=
red not(i = j) implies
    (try(want(s,i),j) =ob= want(try(s,j),i) with (i j)) .
close
--> 1,~i=j,c-want(s,i),~c-try(s,j),~pc(s,j)=wt
open QLOCKobEq
op s : -> Sys .
ops i j : -> Pid .
--
```

```
eq (i = j) = false .
-- eq c-want(s,i) = true .
eq pc(s,i) = rm .
--
eq c-try(s,j) = false .
--
eq (pc(s,j) = wt) = false .
-- |=
red not(i = j) implies
    (try(want(s,i),j) =ob= want(try(s,j),i) with (i j) ) .
close
--> 1,~i=j,~c-want(s,i),c-try(s,j)
open QLOCKobEq
op s : -> Sys .
ops i j : -> Pid .
eq (i = j) = false .
-- eq c-want(s,i) = false .
eq (pc(s,i) = rm) = false .
-- eq c-try(s,j) = true
eq pc(s,j) = wt .
op q : -> Queue .
eq queue(s) = q,j .
-- |=
red not(i = j) implies
    (try(want(s,i),j) =ob= want(try(s,j),i) with (i j) ) .
close
--> 1,~i=j,~c-want(s,i),~c-try(s,j)
open QLOCKobEq
op s : -> Sys .
ops i j : -> Pid .
eq (i = j) = false .
eq c-want(s,i) = false .
eq c-try(s,j) = false .
-- |=
red not(i = j) implies
    (try(want(s,i),j) =ob= want(try(s,j),i) with (i j) ) .
close
```

以上は最初の等式 try(want(S,I),J) = want(try(S,J),I) if not(I = J) の証明譜である．場合分けを行い，全て true を返せば，この等式は成り立つといえる．この証明譜は全て true を返すので，この等式は成り立つ．場合分けは以下の 7 つの場合わけを行っている．

- i=j
- i=j,c-want(s,i),c-try(s,j)
- i=j,c-want(s,i), c-try(s,j),pc(s,j)=wt,queue(s)=empty
- i=j,c-want(s,i), c-try(s,j),pc(s,j)=wt,queue(s)=q,k
- i=j,c-want(s,i), c-try(s,j), pc(s,j)=wt
- i=j, c-want(s,i),c-try(s,j)
- i=j, c-want(s,i), c-try(s,j)

他の等式に関しても場合わけを行って証明譜を作成すれば，全て true を返すことが分かる．よって三つの等式は正当であると考えられる．

モデル検査の実行

等式を導入し，状態を削減した状態からモデル検査にかける．

```
open (QLOCKobEq + QLOCKijkTrans + MEX)
pred mutualEx-ij : Sys .
eq mutualEx-ij(S:Sys) = mutualEx(S,i,j) .
var S : Sys .
vars I J : Pid .
ceq try(want(S,I),J) = want(try(S,J),I) if not(I = J) .
ceq exit(want(S,I),J) = want(exit(S,J),I)
    if (not(I = J) and not(empty?(queue(S)))) .
eq exit(try(want(init,I),I),I) = init .
--> @@@@@ 3 processes @@@@@
red < init > =(*,*)=>* < S:Sys > suchThat (not mutualEx-ij(S)) .
close
```

Search コマンドの前に，等式を導入し全探索を行う．上記の例では，プロセス数 3 つの場合である．いくつかのプロセスに対しモデル検査を行う．実行結果は以下になる．

```
-- opening module QLOCKobEq + QLOCKijkTrans + MEX.. done.__
--> @@@@@ 3 processes @@@@@*
-- reduce in %QLOCKobEq + QLOCKijkTrans + MEX :
             ((< init >) = ( * , * ) =>* (< S >)
             suchThat (not mutualEx-ij(S))):Bool

** No more possible transitions.
(false):Bool
(0.000 sec for parse, 6166 rewrites(0.094 sec),
                      9841 matches, 1324 memo hits)
_*
-- opening module QLOCKobEq + QLOCKijklTrans + MEX.. done.__
--> @@@@@ 4 processes @@@@@*
-- reduce in %QLOCKobEq + QLOCKijklTrans + MEX :
             ((< init >) = ( * , * ) =>* (< S >)
             suchThat (not mutualEx-ij(S))):Bool

** No more possible transitions.
(false):Bool
(0.000 sec for parse, 45281 rewrites(0.563 sec),
                      74272 matches, 9797 memo hits)
_*
-- opening module QLOCKobEq + QLOCKijklmTrans + MEX.. done.__
--> @@@@@ 5 processes @@@@@*
-- reduce in %QLOCKobEq + QLOCKijklmTrans + MEX :
              ((< init >) = ( * , * ) =>* (< S >)
              suchThat (not mutualEx-ij(S))):Bool

** No more possible transitions.
(false):Bool
(0.000 sec for parse, 361586 rewrites(7.328 sec),
                      603329 matches, 78630 memo hits)
_*
-- opening module QLOCKobEq + QLOCKijklmnTrans + MEX.. done.__
--> @@@@@ 6 processes @@@@@*
-- reduce in %QLOCKobEq + QLOCKijklmnTrans + MEX :
             ((< init >) = ( * , * ) =>* (< S >)
             suchThat (not mutualEx-ij(S))):Bool
```

```
** No more possible transitions.
(false):Bool
(0.000 sec for parse, 3161041 rewrites(319.485 sec),
                      5332972 matches, 690421 memo hits)
```

以上のようにプロセス数3つ及び4つでは1秒もかからず探索が終わる．プロセス数5つでは7秒かかり，プロセス数6つでは319秒かかる．これ以上プロセス数を増やして全探索を行うと状態が爆発的に多くなり解が出ない．withStateEqと観測等価を用いた方法よりも，プロセス数を増やして検索することが出来，数倍以上の探索時間の縮減が起こったといえる．

3.5 まとめ

本検証法は，Searchコマンドによるモデル検査にかけるため，無限状態を有限状態へと状態を縮退させるために，等式を用いる．その等式はただ単に導入するだけではなく，妥当性を証明譜を用いて証明している．withStateEqと観測等価関数を用いた方法でも同じようにモデル検査ができるが，探索時間が多くかかり，大規模なシステムになればなるほど，解が出ないことが予想される．一方で，withStateEqと観測等価関数を用いた方法は，観測等価関数を定義するだけで容易に実行できるという利点がある．等式による状態の削減を用いた方法は，探索時間の大幅な縮減が期待できるが，等式の発見においては，思考実験を行ったり，モデルの理解に依存するため，難しくなると考えられる．そこで，次の例題において，等式の発見及び等式の分類の分析を行う．

第4章　通信プロトコルの検証

本章では，通信プロトコル SCP に対し，QLOCK に適用した検証法を適用する手順を説明する．QLOCK は相互排除プロトコルに対し，SCP は通信プロトコルである．QLOCK と SCP では異なる性質を持っている．QLOCK では，全体としてループの構造を持つが，SCP は送受信により状態は常に変化していく．また，QLOCK では，待ち行列の操作が不可分なのに対し，SCP は通信の途中に drop というセルの中身が消失するエラーを表す遷移が割り込む．以上のように，QLOCK よりも複雑な構造をもつ SCP にどのように適用していくかを論じる．

4.1　通信プロトコル SCP

通信プロトコル SCP とは，ABP(Alternating Bit Protocol) の簡易版で，ABP が送受信間の通信路で信頼できない待ち行列を用いるのに対し，SCP は送受信間の通信路で信頼できないセルを用いる．

SCP の動作

セルを介して送信者から受信者にパケットを送る仕組みを考える．ただし，セルは完全に信頼できるものではなく，蓄えている情報が消失するかもしれないとする．SCP には二つのセルを用いる．一つは (cell1) は，送信者から受信者に情報を伝えるために，もう一つ (cell2) は受信者から送信者に情報を伝えるために用いる．送信者は BOOL 型の変数 bit1 を持ち，受信者は BOOL 型の変数 bit2 と，送信されたパケットを蓄えておく List を持つ．SCP の動作は以下のようになる．

- send1:

  送信者は，組<bit1,p_i>を cell1 にいれる．ここで，p_i は i 番目に送りたいパケットを示す．

- rec1:

  送信者は，cell2 が空でなければ，その中のビットを取り出し，bit1 と比較する．等しければ何もしない．そうでなければ，bit1 を補完し，i の値をインクリメントする．

- send2:

  受信者は，bit2 を Cell2 に入れる．

- rec2:

  受信者は，cell1 が空でなければ，その中の組を取り出し，組の最初の要素と bit2 を比較する．等しくなければ何もしない．等しければ，組の 2 番目の要素を List に追加し，bit2 を補完する．

- drop1:

  cell1 が空でなければ，cell1 に蓄えてある情報を消失させる．

- drop2:

  cell2 が空でなければ，cell2 に蓄えてある情報を消失させる．

初期状態では，bit1 と bit2 はともに false で，i は 0 で，cell1 と cell2 はともに空で，list は空である．

以下に SCP の流れを図解する．

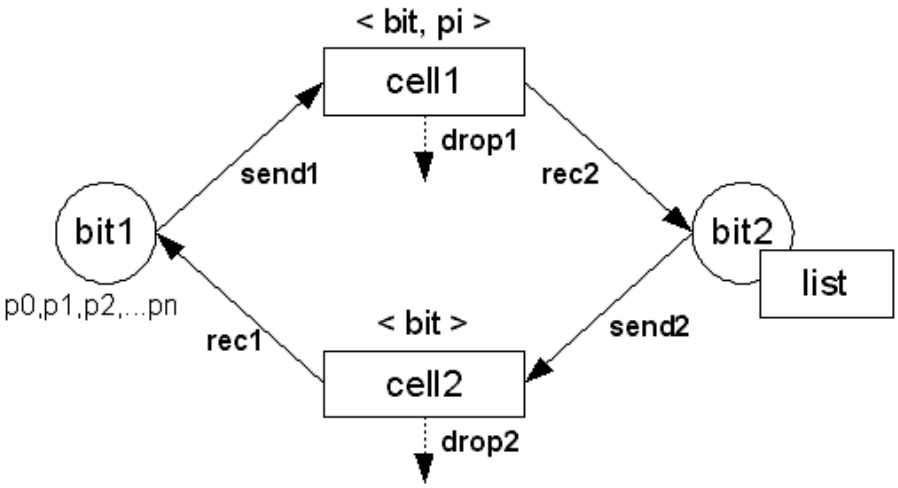

図 4.1: Search コマンドの動作例

4.1.1 モデリング

通信プロトコル SCP を，CafeaOBJ/OTS 法に基づきモデル化を行っていく．OTS とは，初期状態と観測の集合，遷移規則の集合の組によって定義される状態遷移機械である．

観測の定義

SCP の状態を特徴付ける観測値は，SCP の動作から cell1,cell2,bit1,bit2,list,index である．index は，パケットの番号を示す関数である．S_{SCP} の状態を引数とし，これらの値を返す，観測関数を定義する．cell1 は,S_{SCP} の状態を引数に取り，cell1 の中身である組<bit1,p_i>を返す．cell2 は,S_{SCP} の状態を引数に取り，cell2 の中身である bit を返す．bit1 及び bit2 は，S_{SCP} の状態を引数に取り，false か true の bit を返す．list は，S_{SCP} の状態を引数に取り，list 自体を返す．index は，S_{SCP} の状態を引数に取り，状態 S におけるパケットの順番の番号を返す．

よって，S_{SCP} の観測状態の集合 $\mathcal{O}_{SCP}$ は次のような定義になる．

$$\begin{aligned}\mathcal{O}_{SCP} \triangleq \{\ & \text{cell1} : \text{Sys} \to \text{PCell},\ \text{cell2} : \text{Sys} \to \text{BCell},\\ & \text{bit1} : \text{Sys} \to \text{Bool},\ \text{bit2} : \text{Sys} \to \text{Bool},\\ & \text{index} : \text{Sys} \to \text{Nat},\ \text{list} : \text{Sys} \to \text{List}\ \}\end{aligned}$$

Sys,Bool,PCell,BCell,Nat 及び List は，それぞれ状態空間，BOOL 値，自然数と BOOL 値の組のセル，BOOL 値のセル，自然数及び自然数のリストの型である．

遷移の定義

遷移演算は，SCP の動作より，send1,send2,rec1,rec2,drop1,drop2 である．また，それぞれ S_{SCP} の状態のみ引数に取る．

よって，S_{SCP} の遷移の集合 $\mathcal{T}_{SCP}$ は次のような定義になる．

$$\begin{aligned}\mathcal{T}_{SCP} \triangleq \{\ & \text{send1} : \text{Sys} \to \text{Sys},\ \text{send2} : \text{Sys} \to \text{Sys},\\ & \text{rec1} : \text{Sys} \to \text{Sys},\ \text{rec2} : \text{Sys} \to \text{Sys},\\ & \text{drop1} : \text{Sys} \to \text{Sys},\ \text{drop2} : \text{Sys} \to \text{Sys}\ \}\end{aligned}$$

初期状態の定義

初期状態 (init) は,bit1 及び bit2 は false で，cell1 及び cell2 は空になり，list は空 (nil) になり，index は 0 になる．よって，以下のような定義になる．

$$\begin{aligned}\mathcal{I}_{SCP} \triangleq \{\ & \text{init} \mid \text{cell1}(\text{init}) = \text{empty} \wedge \text{cell2}(\text{init}) = \text{empty} \wedge\\ & \text{bit1}(\text{init}) = \text{false} \wedge \text{bit2}(\text{init}) = \text{false} \wedge\\ & \text{list}(\text{init}) = \text{nil} \wedge \text{index}(\text{init}) = 0\ \}\end{aligned}$$

CafeOBJ で記述するにおいて OTS に基づく SCP の振舞の定義（遷移の定義）には，効力条件の定義が必要である．効力条件を示し，遷移を定義することによって，SCP の OTS 仕様は完成する．また，観測の定義は，上記で説明した通りに記述すればよいが，初期状態の定義は等式を用いて記述する必要がある．よって，初期状態の定義と，効力条件を伴った遷移演算の定義を以下に説明する．

初期状態の定義

任意の初期状態を表す定数 init は，以下の通りに定義される．これらの等式は $\mathcal{I}_{QLOCK}$ の定義を CafeOBJ で記述したものに相当する．

```
eq cell1(init) = empty .
eq cell2(init) = empty .
eq bit1(init) = false .
eq bit2(init) = false .
eq list(init) = nil .
eq index(init) = 0 .
```

遷移演算 send1 の定義

遷移 send1 を定義する等式及び効力条件は次の通りの定義される．

```
op c-send1 : Sys -> Bool
eq c-send1(S) = true .
--
ceq cell1(send1(S)) = c(< bit1(S),index(S) >) if c-send1(S) .
ceq cell2(send1(S)) = cell2(S) if c-send1(S) .
ceq bit1(send1(S))  = bit1(S) if c-send1(S) .
ceq bit2(send1(S))  = bit2(S) if c-send1(S) .
ceq index(send1(S)) = index(S) if c-send1(S) .
ceq list(send1(S))  = list(S) if c-send1(S) .
ceq send1(S)        = S if not c-send1(S) .
```

演算 c-send1 は，遷移演算 send1 の効力条件を示している．効力条件は最初の等式で定義されている．続く二つの等式で，条件が満たされた場合，何が起きるかを各観測関数ごとに示している．この場合は，効力条件が true であるので，常に適用される．また，効力条件が true の場合は記述しなくても良い．

遷移演算 rec1 の定義

遷移 rec1 を定義する等式及び効力条件は次の通りの定義される．

```
op c-rec1 : Sys -> Bool
eq c-rec1(S) = not empty?(cell2(S)) .
--
ceq cell1(rec1(S)) = cell1(S) if c-rec1(S) .
ceq cell2(rec1(S)) = empty if c-rec1(S) .
```

```
ceq bit1(rec1(S))  = (if bit1(S) = get(cell2(S))
                     then bit1(S)
                     else get(cell2(S)) fi) if c-rec1(S) .
ceq bit2(rec1(S))  = bit2(S) if c-rec1(S) .
ceq index(rec1(S)) = (if bit1(S) = get(cell2(S))
                     then index(S)
                     else s(index(S)) fi) if c-rec1(S) .
ceq list(rec1(S))  = list(S) if c-rec1(S) .
ceq rec1(S)        = S if not c-rec1(S) .
```

演算 c-rec1 は，遷移演算 rec1 の効力条件を示している．c-rec1 は，cell2 が空でなければ rec1 を実行出来ることを示している．rec1 を実行した後は，cell2 が空になり，cell2 の中身であるビットが受信者のビットと同じであれば bit1 は変わらずパケットの番号を示す index も変化しない，そうでない場合は，bit1 は補完され，index がインクリメントされる．

遷移演算 send2 の定義

遷移 send2 を定義する等式及び効力条件は次の通りの定義される．

```
op c-send2 : Sys -> Bool
eq c-send2(S) = true .
--
ceq cell1(send2(S)) = cell1(S) if c-send2(S) .
ceq cell2(send2(S)) = c(bit2(S)) if c-send2(S) .
ceq bit1(send2(S))  = bit1(S) if c-send2(S) .
ceq bit2(send2(S))  = bit2(S) if c-send2(S) .
ceq index(send2(S)) = index(S) if c-send2(S) .
ceq list(send2(S))  = list(S) if c-send2(S) .
ceq send2(S)        = S if not c-send2(S) .
```

send2 は send1 と同様に定義される．

遷移演算 rec2 の定義

遷移 rec2 を定義する等式及び効力条件は次の通りの定義される．

```
op c-rec2 : Sys -> Bool
eq c-rec2(S) = not empty?(cell1(S)) .
--
```

```
ceq cell1(rec2(S)) = empty if c-rec2(S) .
eq  cell2(rec2(S)) = cell2(S) .
eq  bit1(rec2(S))  = bit1(S) .
ceq bit2(rec2(S))  = (if bit2(S) = fst(get(cell1(S)))
                      then not(bit2(S))
                      else bit2(S) fi) if c-rec2(S) .
eq  index(rec2(S)) = index(S) .
ceq list(rec2(S))  = (if bit2(S) = fst(get(cell1(S)))
                      then (snd(get(cell1(S))) list(S))
                      else list(S) fi) if c-rec2(S) .
ceq rec2(S)        = S if not c-rec2(S) .
```

演算 c-rec2 は，遷移演算 rec2 の効力条件を示している．c-rec2 は，cell1 が空でなければ rec2 を実行出来ることを示している．rec2 を実行した後は，cell1 が空になり，cell1 の中身であるビットが受信者のビットと異なれば，bit2 は変わらず，そうでない場合は，bit2 は補完される．

遷移演算 drop1 の定義

遷移 drop1 を定義する等式及び効力条件は次の通りの定義される．

```
op c-drop1 : Sys -> Bool
eq c-drop1(S) = not empty?(cell1(S)) .
--
ceq cell1(drop1(S)) = empty if c-drop1(S) .
eq  cell2(drop1(S)) = cell2(S) .
eq  bit1(drop1(S))  = bit1(S) .
eq  bit2(drop1(S))  = bit2(S) .
eq  index(drop1(S)) = index(S) .
eq  list(drop1(S))  = list(S) .
ceq drop1(S)        = S if not c-drop1(S) .
```

演算 c-drop1 は，遷移演算 drop1 の効力条件を示している．c-drop1 は，cell1 が空でなければ drop1 を実行出来ることを示している．drop1 を実行した後は，cell1 が空になるり，それ以外の観測は変化しない．

遷移演算 drop2 の定義

遷移 drop2 を定義する等式及び効力条件は次の通りの定義される．

```
op c-drop2 : Sys -> Bool
eq c-drop2(S) = not empty?(cell2(S)) .
--
eq  cell1(drop2(S)) = cell1(S) .
ceq cell2(drop2(S)) = empty if c-drop2(S) .
eq  bit1(drop2(S))  = bit1(S) .
eq  bit2(drop2(S))  = bit2(S) .
eq  index(drop2(S)) = index(S) .
eq  list(drop2(S))  = list(S) .
ceq drop2(S)        = S if not c-drop2(S) .
```

演算 c-drop2 は，遷移演算 drop1 の効力条件を示している．c-drop2 は，cell2 が空でなければ drop2 を実行出来ることを示している．drop2 を実行した後は，cell2 が空になるり，それ以外の観測は変化しない．

通信信頼性

SCP には満たすべき性質として通信信頼性がある．通信信頼性とは，受信者が N 個のパケットを受け取った時，それらは送信者が送った N 個のパケットで，順番を保存している，という性質である．

受信者が受け取ったパケットは list に蓄えられる．このため，このリストを用いて通信信頼性を定式化できる．

```
eq scpInv(S) = (bit1(S) = bit2(S)
               implies (index(S) list(S)) = mk(index(S)))
               and (not(bit1(S) = bit2(S))
               implies list(S) = mk(index(S))) .
```

状態 S までに，受信者が受け取ったパケットの list(S) は，送信者が送信した index(S) 番目までのパケット，もしくは index(S) - 1 番目のパケットまでに送ったパケットであり，受け取った順番と送った順番は一致する．

つまり，受信者がパケットを受け取る時は，過不足無く順番も保存しているということである．

bit1(S) と bit2(S) が同じビットの場合は，送信者が送りたい index(S) 番目のパケットはまだ受信されていない，bit1(S) と bit2(S) が異なる場合は，送信者が送りたい index(S) 番目のパケットは受信されているため，以上のような式になる．mk 関数は自然数 N を引数にとり，N までのリストを作成する関数である．

パケットの有限化

Search コマンドを用いたモデル検査による検証（全探索の検証）を行うためには，到達可能状態の有限化を行わなければならない．到達可能状態が有限であれば，状態空間自体は無限でも全探索のモデル検査が可能になる．したがって，SCP において無限状態をとると考えられる要素はいくつか考えられる．一つはパケットの数を有限化すること，もう一つは受信者のリストを有限化することである．本仕様では，送りたいパケットの数を有限化することにする．

```
mod! NATwNULL {
  [Nat]
  op 0 : -> Nat
  op null : -> Nat
  op s : Nat -> Nat
  op p : Nat -> Nat
  op _=_ : Nat Nat -> Bool {comm}
  vars X Y : Nat
  eq (X = X) = true .
  eq (0 = 0) = true .
  eq (0 = s(X)) = false .
  eq (s(X) = s(Y)) = (X = Y) .
  eq p(s(X)) = X .
  -- null
  eq (0 = null) = false .
  eq (null = null) = true .
  eq s(null) = null .
}
```

モジュール NATwNULL は自然数 (1,2,3,4,...n) に null という定数を追加したものである．null は，パケットの終わりの印を意味し，null は意味の無い空のパケットを示す．例えば，送りたいパケットが 3 つある場合，1,2,3,null となる．また，null の次の順番も null であると定義するため，null を自然数がインクリメントされることはない．

```
mod! NAT-LIST {
  pr(LIST(M <= EQTRIV2NAT))
  op mk : Nat -> List
  var X : Nat
  var L : List
  eq mk(0) = 0 nil .
  eq mk(s(X)) = s(X) mk(X) .
```

```
  eq mk(null) = mk(p(null)) .
  eq null L = L .
}
```

NATwNULL を定義したならば，受信者のリストもそれに対応した形にする．パケットの終わりの null は空パケットのためリストには追加されない．また，mk 関数で null が引数に来た場合は，null の一つ手前の順番までのリストを作成する．

```
eq s(s(s(0))) = null .
```

モデル検査を行う際には，以上の等式を記述する必要がある．以上の式を記述すればパケット数は 3 つに有限化される．0 から始まり，3 が null なので，3 つ（0,1,2）であるまた，これらの式を追加すれば到達可能状態は有限になるが，状態空間自体は無限になる．それは，受信者のリストの長さが無限になっているからである．

4.2 モデル検査への適用

CafeOBJ/OTS 法で記述された仕様を検証する場合には，trans を記述し，Search コマンドを用いて検証する必要がある．遷移規則 (trans) を記述することにより，OTS のダイレクトなモデル検査が可能になる．

モデル検査用の仕様への変換

以下に遷移規則 (trans) の追加の方法を示す．

```
mod! SCPMC2P {
  pr(SCP)

  [ Config ]
  op <_> : Sys -> Config .
  var S : Sys .

  eq s(s(s(0))) = null .

  -- possible transitions in transition rules
  ctrans [send1] : < S > => < send1(S) > if c-send1(S) .
  ctrans [rec2] : < S > => < rec2(S) > if c-rec2(S) .
  ctrans [send2] : < S > => < send2(S) > if c-send2(S) .
  ctrans [rec1] : < S > => < rec1(S) > if c-rec1(S) .
```

```
  ctrans [drop1] : < S > => < drop1(S) > if c-drop1(S) .
  ctrans [drop2] : < S > => < drop2(S) > if c-drop2(S) .
}
```

以上の式は，パケット数が3つのモデル検査用仕様である．pr(SCP) でもともとの CafeOBJ/OTS 法による SCP の仕様を読み込む．SCP の遷移演算は引数が状態 S のみであるため，QLOCK とは違い，プロセスの数ごとに宣言しなくても良い．

モデル検査用の仕様に変換した後，Search コマンドを用いて検証を行う．Search コマンドを用いてモデル検査を行うためには，以下のように行う．

モデル検査用の仕様

```
open (SCPMC + SCPINV)
red < init > =(*,4)=>* < S:Sys > suchThat (not scpInv(S)) .
red < init > =(*,5)=>* < S:Sys > suchThat (not scpInv(S)) .
red < init > =(*,6)=>* < S:Sys > suchThat (not scpInv(S)) .
close
```

SCPMC はモデル検査用の仕様になり，SCPINV は通信信頼性が記述されている．また，SCPINV の仕様は以下のようになる．

```
mod SCPINV {
  pr(SCP)
  pred scpInv : Sys .
  var S : Sys .
  eq scpInv(S) = (bit1(S) = bit2(S)
                  implies (index(S) list(S)) = mk(index(S)))
                  and (not(bit1(S) = bit2(S))
                  implies list(S) = mk(index(S))) .
}
```

モデル検査の実行

Search コマンドを用いたモデル検査の実行の様子を以下に示す．

```
-- reduce in %SCPobEq + SCPMC + SCPINV :
             ((< init >) = ( * , 4 ) =>* (< S >)
             suchThat (not scpInv(S))):Bool
-- reached to the specified search depth 4.
(false):Bool
```

```
(0.000 sec for parse, 46484 rewrites(0.297 sec),
                      89359 matches, 324 memo hits)
-- reduce in %SCPobEq + SCPMC + SCPINV :
             ((< init >) = ( * , 5 ) =>* (< S >)
             suchThat (not scpInv(S))):Bool
-- reached to the specified search depth 5.
(false):Bool
(0.000 sec for parse, 241575 rewrites(1.516 sec),
                      476460 matches, 1208 memo hits)
-- reduce in %SCPobEq + SCPMC + SCPINV :
             ((< init >) = ( * , 6 ) =>* (< S >)
             suchThat (not scpInv(S))):Bool
-- reached to the specified search depth 6.
(false):Bool
(0.000 sec for parse, 1276507 rewrites(8.531 sec),
                      2566992 matches, 4908 memo hits)
```

以上は，パケット数 2 つの仕様に関して実行を行った結果である．深さを一つずつ増やして探索している．全て False の結果を返している．それは，反例が発見されなかった，つまり通信信頼性を満たさない状態は深さ n までは無かったということを示している．

SCP ではパケットの数を増やしても深さが同じならば，同じ探索時間である．それは演算規則の数がパケットの数に依存せず一定だからである．QLOCK はプロセスの数に依存する形になる．深さ 4 では 0.2 秒かかり，深さ 5 では 1.5 秒かかる．また，深さ 6 では，8.5 秒かかることになる．QLOCK に比べて比較的緩やかな時間のかかり方といえる．また，QLOCK と同様にこのままでは全探索はできない．

4.3 withStateEq と観測等価

withStateEq を用いて，ある述語において，新しく検索した項が以前に検索した項と同等であると振舞える．同じ状態を表現する項を有限化するためには，ある述語に対し，項が違えども同じ状態であることを示す述語を作る必要がある．

観測等価関数の定義

QLOCK と同様に withStateEq も用いるためには観測等価関数を作成する必要がある．SCP の観測関数は bit1,bit2,cell1,cell2,list 及び index であり，これらが同じ値を返すかどうかを判断する関数を作ればよい．以下に SCP の観測等価関数を示す．

```
mod SCPobEq {
```

```
  pr(SCP)
  pred (_=ob=_) : Sys Sys .

  vars S1 S2 : Sys .
  eq (S1 =ob= S2) = (cell1(S1) = cell1(S2)) and
                    (cell2(S1) = cell2(S2)) and
                    (index(S1) = index(S2)) and
                    (list(S1) = list(S2)) and
                    (bit1(S1) = bit1(S2)) and
                    (bit2(S1) = bit2(S2)) .
}
```

全ての観測関数において等しいことを示し，それらを and でつなげばよい．list に関しては，list の等価を示す公理が必要であり，これらはモジュール list で定義する．list の等価は list の要素を一つずつ比較する必要がある．以下に list モジュールを示す．

```
mod! LIST (M :: EQTRIV) {
  [List]
  op nil : -> List
  op __ : Elt.M List -> List
  op hd : List -> Elt.M
  op tl : List -> List
  op _=_ : List List -> Bool {comm}

  vars L L1 L2 : List
  vars X Y : Elt.M

  eq hd(X L) = X .
  eq tl(X L) = L .
  eq (L = L) = true .
  eq (nil = nil) = true .
  eq (nil = X L) = false .
  eq (X L1 = Y L2) = (X = Y and L1 = L2) .
}
```

withStateEq と観測等価を用いたモデル検査の実行

withStateEq と観測等価関数を用いたモデル検査の実行結果用のコードを以下に示す．SCP は深さを指定すればパケット数に依存しない．ここではパケット数 3 で定義している．

```
pred obcfg : Config Config .
vars S1 S2 : Sys .
eq obcfg(< S1 >, < S2 >) = (S1 =ob= S2) .

red < init > =(*,9)=>* < S:Sys >
              suchThat (not scpInv(S))
              withStateEq (obcfg(C1:Config, C2:Config)) .
red < init > =(*,10)=>* < S:Sys >
              suchThat (not scpInv(S))
              withStateEq (obcfg(C1:Config, C2:Config)) .
red < init > =(*,11)=>* < S:Sys >
              suchThat (not scpInv(S))
              withStateEq (obcfg(C1:Config, C2:Config)) .
red < init > =(*,*)=>* < S:Sys >
              suchThat (not scpInv(S))
              withStateEq (obcfg(C1:Config, C2:Config)) .

-- reduce in %SCPobEq + SCPMC + SCPINV :
             ((< init >) = ( * , 9 ) =>* (< S >)
             suchThat (not scpInv(S)) withStateEq obcfg(C1,C2)):Bool
-- reached to the specified search depth 9.
(false):Bool
(0.000 sec for parse, 5097572 rewrites(19.375 sec),
                      8644326 matches, 48 memo hits)
-- reduce in %SCPobEq + SCPMC + SCPINV :
             ((< init >) = ( * , 10 ) =>* (< S >)
             suchThat (not scpInv(S)) withStateEq obcfg(C1,C2)):Bool
-- reached to the specified search depth 10.
(false):Bool
(0.000 sec for parse, 10441085 rewrites(39.297 sec),
                      17686886 matches, 52 memo hits)
-- reduce in %SCPobEq + SCPMC2P + SCPINV :
             ((< init >) = ( * , 11 ) =>* (< S >)
             suchThat (not scpInv(S)) withStateEq obcfg(C1,C2)):Bool
-- reached to the specified search depth 11.
(false):Bool
(0.000 sec for parse, 21621547 rewrites(80.156 sec),
                      36602792 matches, 58 memo hits)
```

深さ 9 で約 19 秒かかり，深さ 10 で約 40 秒である．全探索を試みたが，メモリ領域不足などによって解が出てこない．このように withStateEq を用いても解が出てこないことがある．こういった場合には，等式によって状態を削減する方法をとる必要がある．

4.4 等式による状態の削減

同じ状態を表す項を有限化するために，等式を追加することによって，項の表現の有限化を目指す．つまり，ここで追加する等式は，ある遷移演算のパターンに対し，それとは異なる様々な遷移演算を適用しても同じ状態になることを示す等式である．

等式は eq を用いて宣言し，遷移演算の適用パターンのみ記述する．観測関数は用いない．その等式の意味は，左辺の遷移のパターンを適用しても右辺の遷移パターンを適用した状態と同じであり，左辺の遷移パターンは右辺の遷移パターンに常に書き換えられる．つまり，同じ状態を表現する項を有限化できると考えられる．

SCP では QLOCK よりも遷移演算が多く定義され，また SCP の動作も複雑なため，それにより等式の発見も難しくなる．そこで，等式の発見の方法と等式の分類を分析し，体系化したものを以下に示す．それにより，より複雑な使用における等式の発見も容易になると考えられる．

等式の発見

等式の発見は，主にモデルの理解による思考実験で発見する場合が多いと考えられる．しかし，その方法はモデルの理解度に依存し，複雑な等式を発見するのに時間がかかると考えられる．そこで，その方法の補助として，Search コマンドによる実行結果から推敲する方法が考えられる．この方法は，モデルの理解による思考実験よりも比較的機械的に発見することが出来る．しかし，一つ一つ等式を実行結果から考えていくと，等式の数が多くなってしまう可能性がある．

どちらの方法のみを使っても等式の発見は可能であると考えられるが，モデルの検証というのは，モデルの理解を深めるという目的もあるため，モデルの理解による思考実験の補助として，実行結果から推敲する方法をとるべきであると考えられる．

- モデルの理解による思考実験からの発見

  モデルの動作の正規系を念頭に置き，モデルの思考実験により等式を発見する．複雑な等式を発見するのは難しく，理解度に依存する．

- モデル検査の実行結果からの発見

  モデルの動作の正規系を念頭に置き，Search コマンドにより実行結果を分析し発見する．複雑な等式を発見できる可能性がある．しかし，等式の数が多くなる可能性がある．

モデルの動作の正規系とは，モデルの動作の正しい流れを示す．SCP では drop という関数がエラーを示す関数となるので，drop を上手く排除していくように等式を考えると有限化が行いやすいと考えられる．

等式の分類

発見された等式にはいくつかの分類ができる．ここでは，SCP の等式の発見に基づき，いくつかの分類化と分析を行う．以下に示す分類を意識すれば，発見の手助けになると考えられる．発見の仕方は上記で記述した手法をとる．

独立した遷移同士の組み合わせ

独立というのは，ある異なる遷移演算において，それぞれの事前条件が事後状態に干渉しない，また事後状態が事前条件に干渉しないということを示す．CafeOBJ/OTS 法においては事前条件は効力条件のことを示し，事後状態は効力条件が成立した後の観測関数の変化のことを示す．

SCP では，send1 と send2 の効力条件が常に成立するため，他の遷移の事後状態に干渉しない．このことから send1 と send2 は組み合わせられると考えられる．同様に，drop1,drop2 もそれぞれ独立しており，send1,send2 と組み合わせられる．以上のことから，次のような等式を発見できる．

- eq send1(send2(S)) = send2(send1(S)) .
- eq send1(drop2(S)) = drop2(send1(S)) .
- eq send2(drop1(S)) = drop1(send2(S)) .
- eq rec1(drop1(S)) = drop1(rec1(S)) .
- eq rec2(drop2(S)) = drop2(rec2(S)) .

事後状態に注目した遷移同士の組み合わせ

事後状態に注目した遷移の組み合わせのパターンを説明する．rec1,rec2 では各セルの中身のビットを受信するが補完しない場合，bit1,bit2 に変化は起こらず，ただ単に，セルの中身を空にするだけである．それはつまり drop1,drop2 に相当する．このように，事後状態に注目し，それぞれ異なる遷移に置き換える組み合わせがある．

- ceq rec2(S) = drop1(S) if (not(bit2(S) = fst(get(cell1(S))))) .
- ceq rec1(S) = drop2(S) if (bit1(S) = get(cell2(S))) .

if 条件文には，rec1,rec2 における bit1,bit2 が補完しない場合を記述する．

遷移の簡略化に注目した組み合わせ 1

いくつかの遷移が行った状態を表す項が，より簡略された状態を表す項に簡潔化するパターンがある．send1 は連続で複数回遷移がおこるが，それは一回の send1 の遷移と同じである．また，drop1 の後に send1 を行う遷移は，ただ単に send1 の一回の遷移と同じである．以上のように，複数回の遷移をより少ない回数の遷移になるような状態を意識し等式を考える．

- eq send1(send1(S)) = send1(S) .
- eq send2(send2(S)) = send2(S) .
- eq send1(drop1(S)) = send1(S) .
- eq send2(drop2(S)) = send2(S) .

このパターンの特徴は左辺の項の長さが，右辺の項の長さよりも長くなることである．同様に条件付等式も考えることが出来る．

セルの中身が空の時に，send1 を行った後に drop1 を行った状態は，セルは空であり，なにも遷移を行わなかった状態と同じであると考えられる．セルの中身が何か入っていた場合に，send1 を行った後に drop1 を行った状態は，単に drop1 が起こりセルの中身が消失したと同じであると考えられる．以上より，遷移が簡略化するが，セルの中身によって別の状態に簡略化される場合は，if 条件文で別々に定義することが必要になる．

- ceq drop1(send1(S)) = S if empty?(cell1(S)) .
- ceq drop1(send1(S)) = drop1(S) if not(empty?(cell1(S))) .
- ceq drop2(send2(S)) = S if empty?(cell2(S)) .
- ceq drop2(send2(S)) = drop2(S) if not(empty?(cell2(S))) .

遷移の簡略化に注目した組み合わせ 2

遷移の簡略化においては複雑な等式が発見される可能性がある．そういった等式は思考実験によって単に思い付くことが難しい．そこで，モデル検査の実行結果から推測して発見することが薦められる．以下の等式は，今までに発見した等式を加えモデル検査を行い，実験した結果から推測したものである．

- eq send1(rec1(send2(send1(S)))) = send1(rec1(send2(S))) .
- eq send2(rec2(send2(send1(S)))) = send2(rec2(send1(S))) .

send1 及び send2 の上書き除去を示している．モデル検査の実行結果では，上の等式の左辺の項がいくつか発見される．正規系をイメージして考えると，左辺の式における最初に起こる send1 が不必要になることがわかる．send1 の遷移後，rec1,send1 と遷移が起こった場合，最初に起こる send1 は，セルの中身が上書きされるため，遷移が起こらなかったことと同じである．send2 に関しても同様に定義できる．このように，モデル検査の実行結果から推敲することで，あらたな等式を発見することが出来る．

遷移の繰り返しに注目した組み合わせ

SCP の全体としての動作にはループの構造を持っていない．しかし，パケット数を有限化したことにより，最後のパケット (null) において繰り返しの構造を持つようになる．本来，受信者のリストは送信されたパケットにより常に変化していくが，最後のパケット (null) は受信者のリストの格納されないため，index が null の場合，状態の凍結が起こる．このことから，null において，遷移の正規系である send1,rec2,send2,rec1 が二週した状態は，何も起こらなかった状態をと同じであることが言える．bit1,bit2 が T と F の場合があるため二週の繰り返しが必要である．また，条件として，セルの中身が空でなければならない．このように考え導いた等式が以下である．

- ceq rec1(send2(rec2(send1(rec1(send2(rec2(send1(S)))))))) = S
  if (index(S) = null) and empty?(cell1(S)) and
  empty?(cell2(S)) and (bit1(S) = bit2(S)) .

等式の最適化

以上が等式についての発見と分類である．等式の発見は，正規系を意識するとあるが，具体的には drop 関数を排除するように意識して等式を考え出した．drop 関数はエラーを示す関数であるので，動作の正規系には drop 関数を適用した項が出現しない．従って，drop 関数が出現しないように，いくつかの等式を導いた．等式に関しては以上に導いた以上にもっとたくさんある．例えば rec1(rec2(S)) = rec2(rec1(S)), drop1(drop2(S)) = drop2(drop1(S)) などである．しかし，それらの等式を導入すると逆に書換回数が多くなり，モデル検査にかかる時間が多くなってしまう．このように，等式を発見しただけではなく，どの等式を用いるのかという最適化が必要である．等式の最適化については，実行結果の時間を見て，どの等式を導入するのかを考えるのが一番容易だが，いままでに導入した等式から，余計な書換が起こらないかどうかを思考することで，最適化を考えることもできる．

等式の証明

等式の正当性は，証明譜を行って証明する．また，証明譜によって証明を出来る形に等式を変形する必要がある．変形の方法は，QLOCK の章で示している．以下にその例を示す.

```
--> ====================================================
--> send1(send2(S)) = send2(send1(S))
--> ====================================================
open SCPobEq
op s : -> Sys .
-- |=
red send1(send2(s)) =ob= send2(send1(s)) .
close
```

send1(send2(S)) = send2(send1(S)) の証明譜である．場合わけを行わなくても，true を返すので，この等式は正当であるといえる．独立した等式を組み合わせた分類にある等式は比較的証明が容易である．複雑な等式に関しても，数個の場合わけによって証明譜は完成する．複雑な等式でも，まず最初に観測等価関数が適用されて展開される，それにより，場合わけは機械的に行うことができ，無限状態を証明する証明譜よりも複雑になることは無いと考えられる．以下に二つの証明譜の検証の例を示す.

```
--> ====================================================
--> send1(send2(S)) = send2(send1(S))
--> ====================================================
open SCPobEq
op s : -> Sys .
-- |=
red send1(send2(s)) =ob= send2(send1(s)) .
close

--> ====================================================
--> send1(drop2(S)) = drop2(send1(S))
--> ====================================================
-- case
--   /empty?(cell1(s))
--   /~empty?(cell1(s))
-- ----------------------------------------------------
open SCPobEq
op s : -> Sys .
```

```
eq empty?(cell1(s)) = true .
-- |=
red send2(drop1(s)) =ob= drop1(send2(s)) .
close

open SCPobEq
op s : -> Sys .
eq empty?(cell1(s)) = false .
-- |=
red send2(drop1(s)) =ob= drop1(send2(s)) .
close

--> ====================================================
--> rec1(send2(rec2(send1(rec1(send2(rec2(send1(S))))))) = S
-->    if (index(s) = null) and empty?(cell1(s))
-->    and empty?(cell2(s)) and (bit1(s) = bit2(s))
--> ====================================================
-- case
--   /index(s) = null, empty?(cell1(s)), empty?(cell2(s)),
                       bit1(s) = bit2(s), bit2(s) = true
--   /index(s) = null, empty?(cell1(s)), empty?(cell2(s)),
                       bit1(s) = bit2(s), bit2(s) = false
--   /index(s) = null, empty?(cell1(s)), ~empty?(cell2(s)),
                       ~(bit1(s) = bit2(s)
--   /index(s) = null, empty?(cell1(s)), ~empty?(cell2(s))
--   /index(s) = null, ~empty?(cell1(s))
--   /~(index(s) = null)
-- ----------------------------------------------------
open SCPobEq
op s : -> Sys .
eq index(s) = null .
-- eq empty?(cell1(s)) = true .
eq cell1(s) = empty .
-- eq empty?(cell2(s)) = true .
eq cell2(s) = empty .
eq bit1(s) = bit2(s) .
eq bit2(s) = true .
-- |=
red (index(s) = null) and empty?(cell1(s))
```

```
    and empty?(cell2(s)) and (bit1(s) = bit2(s))
    implies rec1(send2(rec2(send1(rec1(send2(rec2(send1(s))))))))
    =ob= s .
close

open SCPobEq
op s : -> Sys .
eq index(s) = null .
-- eq empty?(cell1(s)) = true .
eq cell1(s) = empty .
-- eq empty?(cell2(s)) = true .
eq cell2(s) = empty .
eq bit1(s) = bit2(s) .
eq bit2(s) = false .
-- |=
red (index(s) = null) and empty?(cell1(s))
    and empty?(cell2(s)) and (bit1(s) = bit2(s))
    implies rec1(send2(rec2(send1(rec1(send2(rec2(send1(s))))))))
    =ob= s .
close

open SCPobEq
op s : -> Sys .
eq index(s) = null .
-- eq empty?(cell1(s)) = true .
eq cell1(s) = empty .
-- eq empty?(cell2(s)) = true .
eq cell2(s) = empty .
eq (bit1(s) = bit2(s)) = false .
-- |=
red (index(s) = null) and empty?(cell1(s))
    and empty?(cell2(s)) and (bit1(s) = bit2(s))
    implies rec1(send2(rec2(send1(rec1(send2(rec2(send1(s))))))))
    =ob= s .
close

open SCPobEq
op s : -> Sys .
```

```
eq index(s) = null .
eq empty?(cell1(s)) = true .
eq empty?(cell2(s)) = false .
-- |=
red (index(s) = null) and empty?(cell1(s))
    and empty?(cell2(s)) and (bit1(s) = bit2(s))
    implies rec1(send2(rec2(send1(rec1(send2(rec2(send1(s))))))))
    =ob= s .
close

open SCPobEq
op s : -> Sys .
eq index(s) = null .
eq empty?(cell1(s)) = false .
-- |=
red (index(s) = null) and empty?(cell1(s))
    and empty?(cell2(s)) and (bit1(s) = bit2(s))
    implies rec1(send2(rec2(send1(rec1(send2(rec2(send1(s))))))))
    =ob= s .
close

open SCPobEq
op s : -> Sys .
eq (index(s) = null) = false .
-- |=
red (index(s) = null) and empty?(cell1(s))
    and empty?(cell2(s)) and (bit1(s) = bit2(s))
    implies rec1(send2(rec2(send1(rec1(send2(rec2(send1(s))))))))
    =ob= s .
close
```

モデル検査の実行

以下に，等式を導入して状態の有限化を行った状態でのモデル検査の実行を示す.

```
var S : Sys .
eq send1(send2(S)) = send2(send1(S)) .
eq send1(send1(S)) = send1(S) .
eq send2(send2(S)) = send2(S) .
```

```
eq send1(drop1(S)) = send1(S) .
eq send2(drop2(S)) = send2(S) .
eq send1(drop2(S)) = drop2(send1(S)) .
eq send2(drop1(S)) = drop1(send2(S)) .
eq rec1(drop1(S)) = drop1(rec1(S)) .
eq rec2(drop2(S)) = drop2(rec2(S)) .
ceq rec2(S) = drop1(S) if (not(bit2(S) = fst(get(cell1(S))))) .
ceq rec1(S) = drop2(S) if (bit1(S) = get(cell2(S))) .
ceq drop1(send1(S)) = S if empty?(cell1(S)) .
ceq drop2(send2(S)) = S if empty?(cell2(S)) .
ceq drop1(send1(S)) = drop1(S) if not(empty?(cell1(S))) .
ceq drop2(send2(S)) = drop2(S) if not(empty?(cell2(S))) .
eq send1(rec1(send2(send1(S)))) = send1(rec1(send2(S))) .
eq send2(rec2(send2(send1(S)))) = send2(rec2(send1(S))) .
ceq rec1(send2(rec2(send1(rec1(send2(rec2(send1(S)))))))) = S
    if (index(S) = null) and empty?(cell1(S)) and
    empty?(cell2(S)) and (bit1(S) = bit2(S)) .

-- search
red < init > =(*,11)=>* < S:Sys > suchThat (not scpInv(S)) .
red < init > =(*,12)=>* < S:Sys > suchThat (not scpInv(S)) .
red < init > =(*,13)=>* < S:Sys > suchThat (not scpInv(S)) .
red < init > =(*,*)=>* < S:Sys > suchThat (not scpInv(S)) .
```

これまでに導入した等式をいれ，モデル検査を実行する．

```
-- reduce in %SCPobEq + SCPMC + SCPINV :
             ((< init >) = ( * , 11 ) =>* (< S >)
             suchThat (not scpInv(S))):Bool
-- reached to the specified search depth 11.
(false):Bool
(0.000 sec for parse, 598805 rewrites(2.282 sec),
                      1014527 matches, 325 memo hits)

-- reduce in %SCPobEq + SCPMC + SCPINV :
             ((< init >) = ( * , 12 ) =>* (< S >)
             suchThat (not scpInv(S))):Bool
-- reached to the specified search depth 12.
(false):Bool
```

```
(0.000 sec for parse, 876698 rewrites(3.281 sec),
                      1482418 matches, 356 memo hits)

-- reduce in %SCPobEq + SCPMC + SCPINV :
             ((< init >) = ( * , 13 ) =>* (< S >)
             suchThat (not scpInv(S))):Bool
(false):Bool
(0.000 sec for parse, 1398236 rewrites(5.265 sec),
                      2359354 matches, 380 memo hits)

-- reduce in %SCPobEq + SCPMC1P + SCPINV :
             ((< init >) = ( * , * ) =>* (< S >)
             suchThat (not scpInv(S))):Bool
** No more possible transitions.
(false):Bool
(0.000 sec for parse, 4380426 rewrites(16.062 sec),
                      7371403 matches, 451 memo hits)

-- reduce in %SCPobEq + SCPMC2P + SCPINV :
             ((< init >) = ( * , * ) =>* (< S >)
             suchThat (not scpInv(S))):Bool
** No more possible transitions.
(false):Bool
(0.000 sec for parse, 65979768 rewrites(243.797 sec),
                      111534548 matches, 601 memo hits)
```

深さ 11 で約 2 秒，深さ 12 で約 3 秒であり，withStateEq と観測等価を用いた方法に比べて数十倍の検索の速度が期待できるといえる．また，全探索においては，パケット数 1 つで 16 秒，パケット 2 つで 243 秒かかる．パケット数 3 つは，メモリ的な問題によって解を出すことが難しい．

4.5 まとめ

通信プロトコル SCP において，演繹的検証法と探索的検証法を組み合わせた検証を示した．QLOCK とは異なる性質や動作を持つため，QLOCK とは異なる結果が出た．SCP は遷移演算の引数が状態 S のみをとるため，パケット数の有限化において，演算がそれらに依存するということはない．従って，深さをしている指定する場合は，パケット数が 3 つでも 4 つでも同じ探索時間がかかる．また，主に等式を導入する際，SCP の場合 18 も

の等式が導入され，そのうちの 14 は有限化に必要なものであり，4 つは検索速度を上げるために必要である．さらに思考していけばよりいくつかの等式が出るが，それらに関しては最適化が必要であると考えられる．このように，QLOCK とは異なる性質によって，検証がより複雑になった．それぞれの具体的性質の違いの考察や検証法の違いの考察は，次章で述べる．

第5章　考察

本研究の目的は，演繹的な検証法と探索的な検証法を上手く組み合わせることにより，従来の検証法よりも効率的な検証方法を提案し，その検証法の効果を示すことである．SCP に対してそれらの検証を適用した．その結果からの考察を以下に述べる．

withStateEq と観測等価関数

withStateEq と観測等価関数を用いた方法は，観測等価関数を定義することにより比較的容易にモデル検査の実行が可能になる．観測等価関数の定義は，CafeOBJ/OTS 法における観測関数を全て等価であるか判断するような関数を作ればよい．観測関数に引数がある場合はそれらにも対応させる必要がある．例えば QLOCK の=ob= with がそれにあたる．withStateEq と観測等価関数を用いた方法は，全探索に時間がかかる．それは withStateEq が新しく検索した項と以前に検索した項が観測等価関数において一致しているかを，全ての検索枝に関して一つずつ比較・検査しているからである．単純に考えて，遷移規則の個数分階乗倍に探索時間がかかるといえる．

この方法を用いて，全探索をより探索時間を縮める方法として考えられるのが，観測等価関数の最適化である．withStateEq の一つずつ比較・検査する機能は CafeOBJ の機能で，より速く検索するには，システム自体を変えないといけないが，観測等価関数はユーザ定義なので，速く検索できる可能性がある．例えば，全ての観測関数を比較せずに，必要最低限な要素だけ比較すれば，等価であることが証明できるような観測等価関数が作成できれば，より速く探索できるようになるのではないかと考えられる．

等式の発見

等式の発見と等式の分類については，QLOCK と SCP の検証における等式から分析されたものである．SCP においては等式は 18 個発見された．等式の最適化を行わないのであれば，20 を越える等式が発見できた．QLOCK においては，待ち行列にプロセスが入ったり出たりするループの構造を持つので，それらを有限化するように，SCP はループ構造を持たないが，SCP の非正規系の演算関数である drop を持つのでそれらが項に表れないように考えていく．どちらもモデルの性質の特徴に注目し等式を発見していった．等式の発見は，思考実験だけでは有限化する等式を導き出すのは難しいといえる．そこで等式の発見の補助として，モデル検査で検索項を出力し，等式を導き出す方法があると考えら

れる．SCP においてモデル検査の実行結果から導かれた等式は次の二つである

- send1(rec1(send2(send1(S)))) = send1(rec1(send2(S)))
- send2(rec2(send2(send1(S)))) = send2(rec2(send1(S)))

この等式は，send1,send2 の遷移の除去を示しているが，これらを思考して導き出すのは難しい．そこで，モデル検査の実験結果から推測を行った．この等式を導入することによって有限化が可能になるため，等式の発見の際に補助としてモデル検査の実行結果を観察することは効果があるといえる．

等式の分類

等式の分類については，分類は SCP を分析した結果分析されたものである．SCP は遷移演算を 6 つもち，そのうちの 2 つは効力条件が常に成り立つものである．効力条件が常に成り立つ遷移演算は，他の遷移演算と容易に組み合わせることができ，その結果，等式も多く存在することが分かる．それにより，いくつかのパターンに分類が可能であった．

- 独立した遷移同士を組み合わせるパターン
- 遷移の事後状態に着目したパターン
- 遷移が簡潔になるパターン
- 遷移の繰り返しに着目したパターン

この分類は，SCP の等式の分類だが，QLOCK に適用しても分類化は可能である．QLOCK の 3 つの等式に関して分類を行うと以下のようになると考えられる．

- 独立した遷移同士を組み合わせるパターン
  ceq try(want(S,I),J) = want(try(S,J),I) if not(I = J)
  ceq exit(want(S,I),J) = want(exit(S,J),I) if (not(I = J) and not(empty?(queue(S))))
- 遷移の事後状態に着目したパターン
  なし
- 遷移が簡潔になるパターン
  なし
- 遷移の繰り返しに着目したパターン
  eq exit(try(want(init,I),I),I) = init .

QLOCK の try,want 遷移は引数にプロセスの ID を取るが，try,want 共にプロセス ID が別々の場合，独立した演算とみなすことができる．want,try,exit に関しては QLOCK のループ構造をしめしており，遷移の繰り返しを表現していることがわかる．よって以上のような分類になる．

等式における条件付等式宣言 ceq を用いた等式の場合，if 条件文は仕様内の遷移演算及びその効力条件に記述されている式を用いることが多いことが分かる．例えば，SCP における，if (not(bit2(S) = fst(get(cell1(S))))) などの条件がそれに当たる．それぞれの等式のパターンに対し，条件付き等式がいくつか見られるが，全て遷移演算の定義内に出現した式である．よって，等式は遷移演算に依存していることが考察できる．

等式の最適化

本研究の等式の発見と等式の分類に基づいて等式を導入していく場合，多くの等式が導かれる．すべての等式を追加し，モデル検査を行った場合と，ある一部分を除いた等式を追加して，モデル検査をおこなった場合では，前者の方が等式の数が多いため，モデル検査の検査時間が短くなると考えられるが，追加する等式によっては，後者のほうが検査時間が短くなる可能性がある．以下の等式は，本研究の手法に基づき導かれた等式であるが，以下の等式を今までに導入した 18 個の等式に更に追加すると，書換回数がより多くかかり検査時間が長くなってしまう．

- drop1(drop2(S)) = drop2(drop1(S))
- rec1(rec2(S)) = rec2(rec1(S))

それぞれ独立した遷移同士を組み合わせるパターンとして，drop1,drop2,rec1,rec2 が導かれる．しかし，rec1,rec2 に関しては，ある条件により drop1,drop2 に書き換えられる等式である，

- rec2(S) = drop1(S) if (not(bit2(S) = fst(get(cell1(S)))))
- rec1(S) = drop2(S) if (bit1(S) = get(cell2(S)))

があるため，rec1 と rec2 を組み合わせた等式を加えると，余計な書換が起こる可能性が考えられる．また，drop1,drop2 に関しては，drop1,drop2 に対し，それぞれの遷移関数を組み合わせた等式を既に導入しており，それは drop2,drop2 の遷移を網羅するものであると考えられる．その結果，drop1,drop2 の組み合わせによる等式を新たに追加すると，余計な書換が起こると考えられる．

- send1(drop1(S)) = send1(S)
- send2(drop2(S)) = send2(S)

- send1(drop2(S)) = drop2(send1(S))
- send2(drop1(S)) = drop1(send2(S))
- rec1(drop1(S)) = drop1(rec1(S))
- rec2(drop2(S)) = drop2(rec2(S))

しかし，常に上記の rec1,rec2,drop1,drop2 の組み合わせの等式が検索時間を遅くするとは限らない．本研究で導入した 18 個の等式とは別の等式の導入の仕方をしたのならば，それらの式は検査時間をより速くするかもしれないと考えられる．以上のことから，単に等式を追加するだけではなく，今までに追加した等式を分析し最適化する必要があることが考察される．

探索速度の比較

SCP において演繹的検証法と探索的検証法を組み合わせた方法を用いた結果，飛躍的に速度が上がったことが分かる．以下にその表を示す．

手法	深さ 10	深さ 11	深さ 12	全探索
withStateEq と観測等価	32.9sec	80.1sec	256.3sec	×
等式による状態の削減	1.8sec	2.2sec	3.2sec	243.7sec

以上の上の表ではパケット数が 2 つの場合を示している．深さを指定する場合，探索時間は本仕様の遷移演算の定義においてはパケットの数に依存しない．比較すると探索速度で約数十倍の違いが見られ，また，withStateEq では探索時間が膨大で解を出すことが出来なかった問題が，等式を用いて状態の削減を行った手法だと，約 240 秒で解が出るようになった．これにより，SCP においても演繹的な検証法と探索的な検証法を組み合わせた検証法の有効性を示すことができた．

第6章　まとめ

本研究では，演繹的な検証法と探索的な検証法の組み合わせについての検証法を提案することを目的とする．代数仕様言語 CafeOBJ には，演繹的と探索的に相当する検証法を同時に持ち，演繹的な検証法と探索的な検証法を組み合わせることが出来る．その手法は，証明譜によって正当性が証明された等式によって状態を抽象化させ，Search コマンドでモデル検査にかける手法である．その手法は，相互排除プロトコルに関して有効性が示されているが，それとは異なる性質を持つプロトコルなどについては有効性が示されていない．また，その手法において，状態を抽象化させる等式は非常に重要な要素であるため，等式の発見法や等式の分類についての分析をより詳しく分析する必要があると考えてられる．そこで本研究は以下の二つのことを行う．

- 通信プロトコル SCP に検証法を適用し有効性を確かめる
- 等式の発見法や等式の分類についての体系化について分析する

通信プロトコル SCP に対して検証法を適用した結果，探索時間の短縮の効果が示すことが出来た．それにより検証法の有効性を高めることが出来た．また，等式の発見法や等式の分類についての体系化については，次のような考察を分析した．等式の発見法は，モデルの理解による思考実験を主に等式を発見していき，その補助としてモデル検査のの実行結果である検索項から推測することができる．以上の発見法を用いて導き出された等式には以下のパターンが考えられた．

- 独立した遷移同士を組み合わせるパターン
- 遷移の事後状態に着目したパターン
- 遷移が簡潔になるパターン
- 遷移の繰り返しに着目したパターン

等式の発見法を分析し，パターンを体系化したことにより，別のプロトコルに対して検証を行う際の補助となると考えられる．また，以上の方法を用いて等式を導入した際の，いくつかの等式が探索時間を増やすことがある問題が発見された．それにより，等式の最適化についての分析も行った．

6.1 今後の課題

今後の課題として，別のプロトコルに対して実験を行いよりデータを収集することが上げられる．等式の発見の分類はより深く分析する必要があり，まだ一般化にいたるまで分析されていない．それは，実験例が足りないと考えられる．具体的な実験例として，本研究ではSCPを用いたが，その発展系としてABP[13]に適用し，どう検証されていくのかを分析する必要がある．相互排除プロトコルや通信プロトコルと異なる性質をもつ，認証プロトコルや暗号プロトコルに対しても実験が望まれる．さらに，大規模な仕様に対し実験を行う必要もある．

また，もう一つの課題として，これらの検証を用いた今後の活用法などを考える必要がある．どういった仕様に対し，本研究の検証法を用いれば効果が出るのか，探索時間を減らせるのか，大規模な仕様の場合一部分に用いれば効果が出る，などである．証明譜の検証をベースにして，モデル検査を補助的に使う方法が効率的なのか，モデル検査をベースにして等式の正当性を証明譜で検証する方が効率的なのか，などの議論は今後の課題であるといえる．これらの課題もやはり別の仕様に対して検証を行い，実験を蓄積していくことが必要であると考えられる．

謝辞

本研究を終始に渡りご指導してくださった二木厚吉先生に深く感謝いたします．有益な助言をしていただいた，Rene VESTERGAARD 先生，緒方和博先生，飯田周作先生，中村正樹先生，千葉勇輝先生に御礼を申し上げます．最後に，公私共々お世話になりました，言語設計学講座の諸氏に御礼を申し上げます．

参考文献

[1] CafeOBJ, http://www.ldl.jaist.ac.jp/cafeobj/

[2] A.Nagakawa, T.Sawada, K.Futatsugi, CafeOBJ User's Manual, http://www.ldl.jaist.ac.jp/cafeobj/doc/

[3] Kokichi Futatsugi, Ogata Kazuhiro, Nakamura Masaki, i613: Formal Methods, 2008, http://www.jaist.ac.jp/ kokichi/class/SinaiaSchoolFVSS0803/

[4] Kokichi Futatsugi, Ogata Kazuhiro, Nakamura Masaki, Sinaia School on Formal Verification of Software Systems, 2008 http://www.jaist.ac.jp/ kokichi/class/SinaiaSchoolFVSS0803/

[5] 中野昌弘, モデル検査を用いたシステム検証の支援に関する研究, 北陸先端科学技術大学院大学修士論文, 2003.

[6] 孔維強, OTS/CafeOBJ から OTS/Maude への仕様変換の研究, 電子情報通信学会技術研究報告. SS, ソフトウェアサイエンス, Vol.106, No.120(20060615) pp. 1-6, 2006.

[7] Kazuhiro Ogata, Kokichi Futatsugi Proof Scores in the OTS/CafeOBJ Method, FMOODS, p170-184, 2003: 1.

[8] Cadence SMV, http://www-2.cs.cmu.edu/ modelcheck/smv.html

[9] Maude, http://maude.cs.uiuc.edu/

[10] ジョン・フィッツジェラルド, ペーター・ゴルム ラーセン, ソフトウェア開発のモデル化技法, 岩波書店, 2003

[11] D. Bjorner, C. B. Jones, The Vienna Development Method: The Meta-Language , Springer, 2007

[12] Razvan Diaconescu, Kokichi Futatsugi, CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification, World Scientific, AMAST Series in Computing 6,1998

[13] Bartett, K.A., Scantlebury, R.A., and Wilkinson, P.T., A note on reliable full-duplex transmission over half-duplex links, Communication of the ACM, Vol.12, p260-261, 1969