修 士 論 文

モデル検査を用いたシステム検証の支援に関する研究

指導教官　二木厚吉 教授

北陸先端科学技術大学院大学
情報科学研究科 情報処理学専攻

中野昌弘

2003 年 2 月 14 日

要旨

本研究の目的は，証明論的手法による検証に対しモデル検査を併用することで検証の支援を行うことである．

証明論的手法による検証が成功しない場合，仕様・性質・検証法のどこに問題があるのか容易に認知できない．仮に検証法に問題があるとした場合，主に必用な補題が不足していることが考えられる．必用な補題を発見し，成り立つことを検証してから仕様に追加する．しかし補題が直接証明可能であるとは限らず，さらに別の補題を必要とすることも多い．このような状況において，問題のある場所が仕様・性質・検証法のどこなのか判断することは非常に困難である．

この問題を解決するために，モデル検査を利用する事を提案する．モデル検査は，与えられた仕様・性質に対し全状態を網羅的に検査することで全自動で検証を行う．仕様・性質中に誤りがある場合は誤りを反例として得ることができる．反例から問題の原因を理解し，仕様・性質を修正することができる．モデル検査を導入することにより，証明論的手法による検証が成功しなかった原因が，仕様・性質，検証法のどちらにあるのか判断する明確な判断基準を得ることができる．証明論的手法による検証を行う前にモデル検査で報告される誤りを全て修正することで，証明論的手法による検証段階では仕様・性質が一定レベルの品質を有することを保証できる．

本研究では，振舞仕様からモデル検査の一つである SMV の仕様への仕様変換器を実装した．これにより共通の振舞仕様に対し，CafeOBJ による証明論的手法と SMV によるモデル検査を実現した．両仕様の表現能力の違いから，任意の振舞仕様を SMV 仕様に変換することはできない．無限状態を持つ振舞仕様をモデル検査器で扱うことのできる有限状態遷移機械に変換するため，制限による有限化を導入した．制限は元の仕様の性質を一部保存しないが，容易に有限状態遷移機械を得ることができる．制限とモデル検査により仕様・性質中の誤りの早期発見を実現した．また変換した仕様から得られた反例が，元の仕様でも反例となる事を証明した．

目 次

図目次

表目次

第 1 章

序論

本章では，はじめに研究の背景・目的について述べ，最後に本論文の構成について述べる．

1.1 本研究の背景・目的

システムの正しさを示すために，形式的に記述した仕様を計算機によって検証させる試みが従来から行われてきた．実際に検証は，航空管制システムや原子力発電管理システム，鉄道システムなどで適用が試みられ，大きな成功を納めている．類似の技術としてテスト技法がある．テスト技法は誤りを発見することができるが，どんなにテストを繰り返してもテスト漏れの可能性が依然として残る．一方，検証は誤りが存在しないことを示すことができるため，より強力である．システムが高度に複雑化していく中で，証明論的手法による検証は十分な記述能力・検証能力を有し，今後もその重要性は変わらないものと思われる．

記述した仕様・性質に対し証明論的手法による検証作業は，簡約を繰り返すことで問題の簡単化やより単純な問題へと分割を行う．検証が成功しない場合，仕様・性質・検証法のどこかに問題があるか，場所を特定することは容易ではない．仮に検証法に問題があるとした場合，主に必用な補題が不足していることが考えられる．このような場合，必用な補題を発見し成り立つことを検証してから証明に使用することができる．しかし補題が公理から直接証明可能であるとは限らず，さらに別の補題を必要とすることも多い．このような状況において，問題のある場所が仕様・性質・検証法のどこにあるのか判断することは非常に困難である．

近年ハードウェアの分野において，入力の全数検査による検証技術であるモデル検査が大きな成功を収め，注目されている．モデル検査は，与えられた仕様・性質に対し全状態を網羅的に検査する．モデル検査的手法による検証は，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はできない．しかし証明論的手法に無い特徴として，検証を完全に自動でできる事，性質が仕様上で成り立たない場合に反例を示すことができる事が挙げられる．しかしモデル検査は全数探索を行うため，無限状態を扱うような仕様を扱うことができない．また現実の問題は非常に規模が大きく，状態爆発を起こし適用できない問題も多い．

本研究の目的は，証明論的手法による検証に対しモデル検査を併用することで検証の支援を行うことである．証明論的手法による検証が成功しなかった原因が仕様・性質にある場合，部分問題に対してモデル検査を適用することにより反例が得られ，仕様を修正できる事が期待される．検証の手順として，

1. 部分問題に対し，モデル検査による検証を行う．モデル検査で報告される誤りを全て修正し，誤りが報告されないようにする．
2. モデル検査で誤りが報告されない場合に証明論的手法により検証を行う．

仕様が修正される時や証明に用いる補題の検証は， 1, 2 の手順により検証を行う．モデル検査で報告される誤りを全て修正することにより，証明論的手法による検証段階では仕様・性質が一定レベルの品質を有することが期待される．

1.2 本稿の構成

本稿では 2 章で証明論的手法とモデル検査的手法について述べ，それぞれの検証手法の利点・欠点について述べる．3 章では本研究で扱う CafeOBJ と SMV について，仕様の文法やその意味などを述べる．4 章では本研究の実装システムについて述べ，証明論的手法とモデル検査的手法を併用した検証手法を提案する．5 章では例題を用いて提案手法の効果を確かめ，6 章では本稿をまとめ，関連研究，今後の課題について述べる．

第 2 章

計算機による証明

計算機による検証に関する研究は古くから行われて来た．証明する対象も数学の定理，ハードウェアの正しさ，ソフトウェアの正しさ，システムの正しさなど様々である．計算機を利用して証明するためには問題を形式的に記述し，計算機に正しく解釈させる必要がある．このような問題の記述を形式的仕様 (Formal Specification) と呼ぶ．計算機を利用し，この形式仕様がある命題を満たすかどうかについて証明する．これらの方法も，多くのものが提案・実装され実際の問題にも応用されている．これらの証明方法は大きく 2 つに分けることができる．

- 形式仕様に対して，演繹的手続きで命題の真偽を導出することで証明する方法.
- 問題空間を機械的に網羅し，全ての場合を検査する方法.

前者に基づく証明法を証明論的手法と呼び，後者に基づく方法を総じてモデル検査的手法と呼ぶ．本章では証明論的手法，モデル検査的手法について述べたあとに，検証する性質について述べる．

2.1 証明論的手法

証明論的手法による検証は，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できる．証明論的手法による検証にはさらに次の 2 つに分類することができる．

証明検証系 (proof checker) は，形式仕様と証明する命題，その証明を与え，指定した規則により証明が演繹的に導いているかどうかを検証する．

定理証明系 (theorem prover) は，形式仕様と命題からその証明を自動生成する．

証明検証系では人間が記述した証明をトレースする．証明中に誤りや論理の飛躍がある場合，ユーザにエラーを報告し修正を促す．そして修正した仕様に対して検証を再度行う．この作業を繰り返し行うことで，正しい証明を構築していく．証明検証系は，公理と推論規則から演繹的に証明していく．定理証明系で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．一方，定理証明系による自動証明は非常に難しく，完全に自動的に証明できる問題は限られている．命題をいくつかの命題に分割して問題を簡単にしたり，補題を順に証明させることで全体の証明を生成することが可能になる．この処理は人が証明の指針を与えることで自動証明の手助けを行う．証明支援系においても，全ての証明が公理と推論規則だけから導出される証明だけでは多大な労力が必要になる．このため既に証明された定理については，他の証明で再利用することができる．また簡単な自動定理証明の機能を備えた処理系も多く，実際の証明検証系と定理証明系は多くの共通する性質を持つ．

証明論的手法による証明では

- ほとんど自明と思われるような補題についてもユーザによって与える．
- 仕様を修正しなければならない理由をシステムから得ることが困難な場合がある．

等の問題点を持つ．人間の直感と形式的な証明の間には大きな差が有る．例えば任意の自然数に対して，加法が対称律 ($\forall a \in Nat, \forall b \in Nat$ に対し $(a + b = b + a)$) を満たすことは直感的に理解できるが，これを証明論的手法を公理から証明するためには 2 つの補題を用いて帰納的に証明しなければならない．

以下では証明論的手法による検証器についていくつか取り上げ，簡単に説明する．

Mizar [**?**] は数学の証明検証を目的とした証明検証系であり，自然語に近い文法で証明スコアを記述することができる．Mizar では，これまでに証明されてきた定理を MML(Mizar Mathmatical Library) として提供しており，ユーザはこれらの定理を自由に利用することができる．

Isabelle/HOL [NPW] は数学の定理証明・ハードウェア検証・ソフトウェア検証など幅広い分野で使用されている．HOL では証明したい性質をゴールと呼ぶ．ゴールを証明するためにいくつかのサブゴールに簡約し，全てのサブゴールの証明を試みる．全てのサブゴールが証明されるとゴールが証明されたことになる．ゴールからサブゴールに簡約するルールをタクティク (tactic) と呼び，これは証明する上で必要となるヒューリスティクスである．繰り返し・選択・逐次などのタクティカル (tactical) と呼ばれるオペレータを用いて複数のタクティクをまとめ，新たなタクティクを作成することもできる．タクティクを追加していくことにより定理証明器としての能力を向上させる．

CafeOBJ [RK98, caf] は代数に基づく検証器である．仕様はシグニチャと等式からなるモジュール単位で記述する．シグニチャは項を構成する定数と演算からなり，等式は項の間の等価関係を記述する．CafeOBJ では証明したい性質を項として表現し，等式を簡約規則として項を繰り返し簡約させる．証明が成功する場合，簡約の結果 true が得られる．true が得られず簡約が途中で停止した場合，必要な補題を等式として追加し簡約を再度試みる．

2.2 モデル検査的手法

モデル検査的手法は，ハードウェア検証で特に大きな成功を収めた検証手法である．最近ではプロトコルの検証やソフトウェアの検証にも広く用いられるようになった．

モデル検査的手法の前身である状態探索系による検証は，状態機械の全ての遷移系列に対してプロパティを網羅的に調べることにより行われる．このため証明論的手法では実現が困難であった自動検証が容易に実現できる．検証した結果性質が満たされるならば true が得られ，満たされないならばその反例を得ることができる．状態探索系としては Murϕ がある．

モデル検査系 [ED99, ea01] は，状態探索系と同じように状態機械の遷移系列を網羅的に検証する．この時，モデル検査系では調べる性質を時相論理 (Temporal Logic) による論理式で与える．時相論理は 1 階の命題論理式に時相オペレータとパス量化子を追加した論理式である．時相オペレータとパス量化子には表2.1，2.2に示すものがある．

時相論理には，線形時間時相論理と分岐時間時相論理の二つがある．それぞれの時相論理について説明した後，モデル検査において非常に重要な技術である BDD について説明する．

2.2.1 分岐時間時相論理

分岐時間時相論理 (Branching Time Temporal Logic) は状態遷移の分岐構造を表現することができ，複数のパスに対する表明を記述することができる．モデル検査系でよく使用される分岐時間時相論理に計算木論理 (Computational Tree Logic, CTL) がある．CTL では表2.1で示した時相論理式に対して，時相オペレータ X, F, G, U, W の直前に必ずパス量化記号 A または E が無ければならない．CTL では結局，表2.3に示すオペレータを使うことができる．

CTL で書かれた性質に対するモデル検査は，状態遷移機械の (状態数 + 遷移経路数) × 性質中の時相オペレータ数 に対して線形時間で検証ができる．また CTL 言語では後述する

表 2.1: 時相オペレータの種類とその意味

時相論理式	時相論理式の意味
X ϕ	次の状態で ϕ が成り立つ.
F ϕ	いつかは ϕ が成り立つ．(F ϕ = ¬ G ¬ϕ)
G ϕ	現在の状態以降の全ての状態で ϕ が成り立つ.
ϕ_1 W ϕ_2	現在の状態から ϕ_2 が成り立つまで ϕ_1 が成り立つ．このとき ϕ_2 は無限に成り立たなくともよい.
ϕ_1 U ϕ_2	現在の状態から ϕ_2 が成り立つまで ϕ_1 が成り立ち，ϕ_2 はいつかは成り立つ. (ϕ_1 U ϕ_2 = (F ϕ_1) ∧ (A ϕ_1 W ϕ_2))

パス量化式	量化記号式の意味
A ϕ	全ての実行系列に対して ϕ が成り立つ. (A ϕ = ¬ E ¬ϕ)
E ϕ	ある実行系列に対して ϕ が成り立つ.

※ ϕ, ϕ_1, ϕ_2 は任意の時相論理式，または命題論理式

表 2.3: 分岐時間時相論理の持つ時相オペレータの種類とその意味

時相論理式	時相論理式の意味
AX ϕ	全ての次の状態で ϕ が成り立つ.
EX ϕ	次の状態で ϕ が成り立つ実行経路が存在する.
AF ϕ	全ての実行経路に対し，いつかは ϕ が成り立つ.
EF ϕ	いつかは ϕ が成り立つ実行経路が存在する.
AG ϕ	全ての実行経路に対し，現在の状態以降の全ての状態で ϕ が成り立つ.
EG ϕ	現在の状態以降の全ての状態で ϕ が成り立つ実行経路が存在する.
A ϕ_1 U ϕ_2	全ての実行経路に対し，現在の状態から ϕ_2 が成り立つまで ϕ_1 が成り立ち，ϕ_2 はいつかは成り立つ.
E ϕ_1 U ϕ_2	現在の状態から ϕ_2 が成り立つまで ϕ_1 が成り立ち，ϕ_2 はいつかは成り立つ実行経路が存在する.
A ϕ_1 W ϕ_2	全ての実行経路に対し，現在の状態から ϕ_2 が成り立つまで ϕ_1 が成り立ち続けるか，ϕ_2 が無限に成り立つ.
E ϕ_1 W ϕ_2	現在の状態から ϕ_2 が成り立つまで ϕ_1 が成り立ちつづけるか，ϕ_2 が無限に成り立つ実行経路が存在する.

※ ϕ, ϕ_1, ϕ_2 は任意の時相論理式，または命題論理式

公平性という概念を記述することができないが，各処理系で計算方法を工夫して対応していることが一般的である．

CTL で記述された性質 ϕ に対するモデル検査を説明する．性質 ϕ の部分項を ψ で表し，ϕ を満たす状態の集合を $Sat(\phi)$ で表す．直前の状態の集合を得る演算 Pre を使用する．

CTL に対するモデル検査は，まず時相オペレータも $\wedge, \vee, \neg$ も含まない基本的な命題 ψ_- について真偽を調べる．各命題 ψ_- が成り立つ状態 $Sat(\psi_-)$ を全ての ϕ の部分項に対して探索した後，各 $Sat(\psi_-)$ を基本項として，表2.4に示す方法によって各時相オペレータを含んだ部分項が計算される．最終的に $Sat(\phi)$ が計算された後，$Sat(\phi)$ の補集合の各状態から状態遷移機械を逆向きに辿っていき，初期状態に到達すると性質 ϕ を満たさない反例が得られる．

表 2.4: CTL オペレータと boolean オペレータに対する計算方法

集 合	計 算 方 法
$Sat(\neg\psi)$	$Sat(\psi)$ の補集合.
$Sat(\psi_1 \wedge \psi_2)$	$Sat(\psi_1) \cap Sat(\psi_2)$
$Sat(\psi_1 \vee \psi_2)$	$Sat(\psi_1) \cup Sat(\psi_2)$
$Sat(EX\psi)$	$Pre(Sat(\psi))$
$Sat(AX\psi)$	$Sat(\neg(Pre(\neg\psi)))$.
$Sat(EF\psi)$	$Sat(\psi)$ から Pre を 0 回以上行って到達できる全状態.
$Sat(AG\psi)$	$Sat(\neg EF\neg\psi)$
$Sat(EG\psi)$	まず 1)$X := Sat(\psi)$. 2) $X := Sat(X \wedge Pre(X))$. 3) X の不動点が得られるまで 2 を繰り返す．X の不動点が $Sat(EG\psi)$ である．
$Sat(AF\psi)$	$Sat(\neg EG\neg\psi)$
$Sat(A\psi_1 U\psi_2)$	$Sat(A\psi_1 U\psi_2) \equiv Sat(\psi_2 \vee (\psi_1 \wedge EXtrue \wedge AX(A\psi_1 U\psi_2)))$ である．1)$X := Sat(\psi_2)$ を求める．2)$X := Sat(X \vee (\psi_1 \wedge Pre(X) \wedge (\neg(Pre(\neg X)))))$ を求める． X の不動点が得られるまで 2 を繰り返す．X の不動点が $Sat(A\psi_1 U\psi_2)$ である．
$Sat(A\psi_1 W\psi_2)$	$Sat(A\psi_1 U(\psi_2 \vee AG\psi_1))$
$Sat(E\psi_1 U\psi_2)$	$Sat(E\psi_1 U\psi_2) \equiv Sat(\psi_2 \vee (\psi_1 \wedge EX(E\psi_1 U\psi_2)))$ である． 1)$X := Sat(\psi_2)$ を求める．2) $X := Sat(X \vee (\psi_1 \wedge Pre(X)))$. 3) X の不動点が求まるまで 2 を繰り返す． X の不動点が $Sat(E\psi_1 U\psi_2)$ である．
$Sat(E\psi_1 W\psi_2)$	$Sat(E\psi_1 U(\psi_2 \vee EG\psi_1))$.

CTL を扱うことができる処理系としては SMV, NuSMV などがある．またこれらの処理系では公平性を仮定した検証を行う事ができる．

2.2.2 命題線形時間時相論理

命題線形時間時相論理 (Propositional Linear-time Temporal Logic, PLTL) は，状態遷移機械が取りうる一つの遷移系列に対する表明を記述する．このため，CTL のように複数のパスに対する表明は記述することができず, 量化記号 A, E は PLTL 言語には現れない．PLTL 言語で使用することができるオペレータは，表2.1で示した時相オペレータであり，任意の組合せで使うことができる．

オートマトン A が PLTL で記述された性質を検証する方法について簡単に述べる．まず検証する性質 ϕ を満たさない時に限り受理されるオートマトン $B_{\neg\phi}$ を作成する．次にオートマトン A と $B_{\neg\phi}$ を合成する．合成したオートマトン上で $B_{\neg\phi}$ の受理状態に対応する状態に到達するならば，A は性質 ϕ を満たさ無いことになる．PLTL に対するモデル検査は，オートマトンが任意の無限長の入力に対し受理される入力があるかどうかという，よく知られた問題に帰着することができる．

PLTL を扱うことができる処理系としては SPIN などがある．

2.2.3 BDD

SMV では有限の状態からなる遷移機械に対し，遷移系列を網羅的に検証する．しかし状態数が増えるにつれすぐに計算量の限界に達してしまい，検証を行う事ができなくなる．これを状態爆発 (State Explosion) という．

モデル検査で扱える状態数を飛躍的に増やした技術として 2 分決定グラフ (Binary Decision Diagram, BDD) [CGL94] がある．BDD は任意の論理式に対応する真理値表を決定木 (Decision Tree) を用いて表現する．例えば，論理式 $P = (\neg x_0 \wedge \neg x_2) \vee (x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge x_2)$ の真理値表は表2.5のようになる．また論理式 P に対応する決定木を図3.5に示す．

図3.5は表2.5に直接対応する決定木となっている．決定木はルートである x_0 から 3 回順にたどることで，ある x に対する P の真偽を計算することができる．一般に，n 個のブール変数に対する決定木は n 回ノードをたどることで求めることができる．しかし，決定木を構築するために必用なノード数は $2^n - 1$ 個である．

図3.5で示した決定木は非常に大きなサイズとなるので，決定木に対して簡約を行う．BDD は簡約した決定木である．決定木簡約は 2 つの操作によって行われる．

1. まず決定木に現れる部分木の共通部分を探しまとめる．図2.2では 3 つの x_2 ノードが 1 つにまとめられ，全ての $\boxed{0}$, $\boxed{1}$ ノードがそれぞれ一つにまとめられている．

表 2.5: 論理式 $P = (\neg x_0 \wedge \neg x_2) \vee (x_1 \wedge \neg x_2) \vee (x_0 \wedge \neg x_1 \wedge x_2)$ に対する真理値表

x_0	x_1	x_2	P
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

2. 次に，ノードの遷移先が 1 と 0 の場合で同じであるノードを削除する．図3.6では 1 つの x_1 ノードが削除されている．

この 2 つの操作を行う事で非常に小さい決定木を得ることができる．

BDD は論理式を非常に効率的に表現することが可能である．x_0, x_1, x_2 という 3 変数からなる状態機械と考えると，BDD によって最適化された論理式 P は，P を満たす状態の集合を効率的に表していることになる．また，BDD は基本的な計算 (積集合，和集合，補集合，集合の大小関係など) を非常に簡単に表現することが可能である．これらの計算は，表2.4でも示したようにモデル検査を行う上でも基本的な計算となっており，BDD を用いてモデル検査を行う効果は非常に大きい．

SMV では Ordered BDD(OBDD)[EOD93] という BDD に基づいて計算を行う．BDD は x_0, x_1, x_2 の順番は変更することができなかったが，OBDD では，全ての部分木で変数順が等しいという条件付ではあるが，順序を変更することができる．BDD は変数の順序に依存して表現の効率が大きく変わるので，SMV では変数順を与えることで検証を効率よく行う事ができる．現在では Free BDD, ADD, Zero-suppressed BDD など非常に多くの BDD が提案され活発に研究されている．

2.3 検証する性質

記述された形式仕様に対して，主に安全性と活性と呼ばれる性質を検証することが一般的である．どちらもシステムの満たすべき重要な性質を表現することができる．

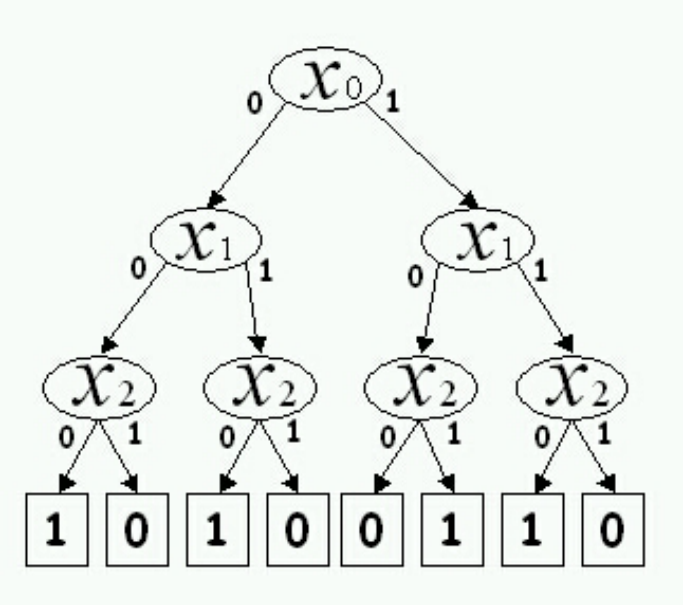

図 2.1: 論理式 P に対する決定木 (最適化無し)

2.3.1 安全性

安全性 (safety property) とは常に成り立つ性質であり，システムに対して最も基本的で重要な性質である．安全性の例として次のようなものがある．

- 相互排他問題に対しては，複数のプロセスが同時に共有資源を獲得することが無いこと．
- 認証プロトコルに対しては，認証相手を誤って認証しないこと．
- Unix のユーザ管理では，正しい root パスワードを入力しない限り root ユーザになれないこと．
- 発射ボタンを押さない限り，ミサイルを発射しないこと．

安全性は，一般的には好ましくない状態に陥ることが無いことを表し，これが安全性と呼ばれる理由である．

安全性は線形時間時相論理, 分岐時間時相論理ではそれぞれ表2.6のように表される．

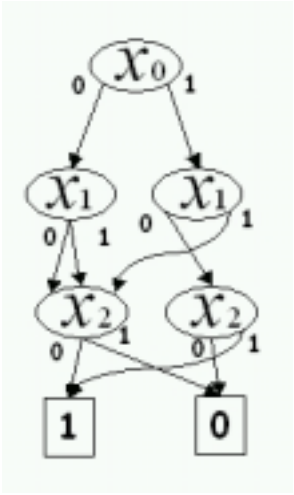

図 2.2: 論理式 P に対する決定木
(最適化途中)

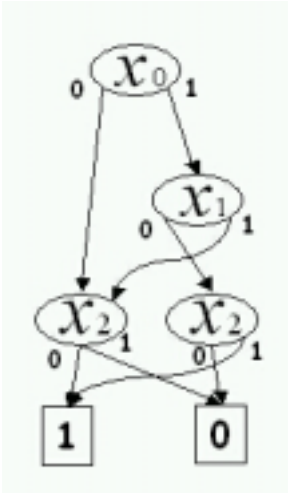

図 2.3: 論理式 P に対する決定木
(最適化後)

表 2.6: 各時相論理での安全性

時相論理の種類	安全性の一般形
分岐時間時相論理	AG ϕ^-, A ϕ_1^- W ϕ_2^-
線形時間時相論理	G ϕ^-, ϕ_1^- W ϕ_2^-

※ $\phi^-, \phi_1^-, \phi_2^-$ は命題論理式，または時相オペレータに E, F, X, U を含まない時相論理式

ある状態機械の全ての状態で性質 ϕ^- が成り立つ時，その性質は安全性である．上で挙げた安全性の例では，上 2 つが G オペレータを使用した安全性の例であり，下 2 つが W オペレータを使用した安全性の例である．

安全性は形式的には以下の様に定義されている．

安全性の定義 P を任意の性質，C を任意の状態遷移機械とし，C が P を満たすとする．
このとき，$\forall C' \sqsubseteq C$ に対して C' が P を満たすならば，P は安全性である．
ここで $C' \sqsubseteq C$ は，C' で現れる任意の実行系列が C の実行系列の一部として現れることを意味する．

この定義から他のオペレータを使用した性質が安全性では無いことを示すことができる．以下では分岐時間時相論理式に対して説明するが，線形時間論理式においても同様に説明できる．

E オペレータ を使用した性質 E P は安全性では無いことを示すため，ある C において P を満たす状態が実行経路に存在したとする．$C' \sqsubseteq C$ として，E P が成り立たない実行経路のみを取った場合，当然のことだが E P が成り立つ実行経路は存在しない．よって定義より安全性ではなく，E を使用した全ての性質は安全性ではない．

F オペレータ を使用した性質 AF P は安全性で無いことを示すため，ある C において，全ての実行経路に対して P を満たす状態が存在するとする．$C' \sqsubseteq C$ として P が成り立つ前までの実行経路を取った場合，F P が成り立つ実行経路は存在しない．よって定義より安全性ではない．

X オペレータ を使用した性質 AX P は安全性で無いことを示すため，ある C において，全ての実行経路に対して次の状態で P を満たすとする．$C' \sqsubseteq C$ として P が成り立つ前までの実行経路を取った場合次の状態が存在しないので，AX P が成り立つ実行経路は存在しない．よって定義より安全性ではない．

U オペレータ を使用した性質 A P_1 U P_2 が安全性で無いことを示すため，ある C において，全ての実行経路に対して P_2 を満たす状態になるまで，P_1 が成り立ちつづけているとする．$C' \sqsubseteq C$ として P_2 が成り立つ前までの実行経路を取った場合，P_2 は成り立つことが無いため A P_1 U P_2 が成り立つ実行経路は存在しない．よって定義より安全性ではない．これは A P_1 U P_2 = (AF P_1) ∧ (A P_1 W P_2) であることからも分かる．

よって X, U オペレータを使用した性質は安全性ではないことが分かる．また，E, F オペレータに対して否定が付かない性質も安全性で無いことが分かる．E, F オペレータに対して否定がつくと，それぞれ頭に否定のつかない A, G オペレータで表現できるため注意が必要である．またこれらの結果から，デッドロックが起こらないという性質 AG EX true は，X オペレータを使用するので安全性ではないことが分かる．

2.3.2 活性

活性 (liveness property) は，いつかは成り立つ性質である．活性の例としては以下のようなものがある．

- 相互排他問題に対して，資源獲得を要求したならばいつかは資源を獲得すること．
- エレベータで目的の階数を入力したならば，いつかはその階でドアが開くこと．

- 信号が赤であるならば，いつかは青になること．
- プログラムがいつかは停止すること．

活性は，一般的にはいつかはよい事が起こる事と言うことができる．構文的には，分岐時間時相論理・線形時間時相論理で書かれた性質が活性であるかどうかは次のように判定できる．

性質 P に時相オペレータ F または U を含むとき，それは活性である．

哲学者の食事問題 (Dining Philosophers Problem) や相互排他問題 (Mutual Exclusion Problem) など，複数の哲学者プロセスでものを共有するような問題において活性は特に重要な性質を記述することができる．哲学者が食事をしようとした時にいつかは食べることができる事や，あるプロセスが資源の獲得要求を出すといつかは獲得することができる事などは，上述の例と同様に活性で表現できる．しかしある哲学者・プロセスが独占的にフォーク・共有資源を使用することで他の哲学者・プロセスが食事をとる・資源を獲得することができない状況がある．この状況を飢餓 (Starvation) と呼ぶ．PLTL ではこのような独占的に使用することは無いという仮定を記述することができる．この仮定を公平性 (Fairness) と呼ぶ．公平性は，さらに弱公平性と強公平性とに分けられる．

弱公平性 (Weak Fairness) を仮定することにより，哲学者がフォークを取る・プロセスが資源獲得を試みる等のチャンスが，全ての哲学者・プロセスに対して公平に与えられる．

強公平性 (Strong Fairness) を仮定することにより，全ての哲学者・プロセスに対して，フォークを取ることができる条件・資源を獲得することができる条件を満たす時に，哲学者がフォークを取る・プロセスが資源獲得を試みる動作が，全ての哲学者・プロセスに対して公平に与えられる．

強公平性は，弱公平性よりも強い仮定である．強公平性の元である活性 P が成り立ったとしても，弱公平性の元で P が成り立つとは限らない．しかしその逆で，弱公平性の元で活性 P が成り立つならば，強公平性の元でも P は成り立つ．

第 3章

CafeOBJ言語とSMV言語

ここでは本研究で扱う CafeOBJ 言語と SMV 言語について説明する．

3.1 CafeOBJ仕様

CafeOBJ 言語では代数仕様 (Algebraric Specification) と呼ばれる仕様を記述する．代数仕様はソート (Sort) とソートの上での演算 (Operation) からなる指標 (signature) により言語を定義し，指標によって作られるの 2 項の間の等価関係を等式よって定義する．CafeOBJ 言語では指標と等式から構成される Module を単位として仕様を記述する．

CafeOBJ 言語のモジュール宣言を例を用いて説明する．ここでは図3.1に示す Peano の自然数を用いて説明する．

```
 1: mod! NAT{                      --  コメント
 2:   [ Zero NzNat < Nat ]         -- ソートの宣言
 3:   op 0 : -> Zero               -- 演算の宣言
 4:   op s : Nat -> NzNat          -- 演算の宣言
 5:   op _ + _ : Nat Nat -> Nat    -- 演算の宣言
 6:
 7:   vars P Q : Nat
 8:   eq 0 + P = P .               -- 等式の宣言
 9:   eq s P + Q = s (P + Q) .     -- 等式の宣言
10: }
```

図 3.1: Peano の自然数を表す NAT モジュール

Peano の自然数というのは一つの定数 0 と後者関数 s によって自然数全体を定義する．ここでは自然数上での加算を定義している．まず **mod!** MODULENAME{ ‥‥ } により MODULENAME という名前をもつモジュールの定義が開始され，“**{**” と “**}**” によって囲まれた

部分でモジュールの指標と等式が定義される．よってここではNATという名前をもつモジュールの指標と等式の定義が開始される．2行目から5行目までが指標部である．まず2行目でソートの宣言が行われている．ソート宣言は“[”と“]”で囲んで記述する．ここでは Zero NzNat Nat の3つのソートが宣言されている．NzNat と Nat の間にある“**<**”はソート間の順序関係を記述する．これにより，Zero というソートを持つ全ての項と NzNat というソートを持つ全ての項は，Nat の項でもあることが宣言されている．これが順序ソート (Order Sort) である．順序ソートを導入することにより演算のオーバーローディングが明確に定義される．ここでは各ソートは次のような意味を持っている．

Zero はゼロを表すソートである．要素として0のみを持つ．

NzNat は非ゼロの自然数 (Non-zero Nat) を表すソートである．要素として1以上の任意の自然数を持つ．

Nat は自然数全体を表すソートである．要素として0以上の任意の自然数を持つ．

実際には，ソートのこうした意味は等式によって定義される．

3 – 5行目では演算を定義している． 演算の宣言は次のような構文を持つ．

“**op**” 演算名 “**:**” アリティ “**–>**” コアリティ

演算が引数にとるソートの組をアリティ(arity) と呼び，演算が持つソートをコアリティ (coarity) と呼ぶ．3行目ではアリティに何もとらず，コアリティに Zero を持つ演算0が定義されている．0はアリティに何も持たないため定数である．4行目ではアリティに Nat を一つとり，NzNat を返す演算 s を定義している．この演算の意味は自然数を一つもらい，それに1を足した自然数を返す．このためアリティが Nat，コアリティが NzNat となる．この関数を後者関数 (successor 関数) と呼ぶ．最後に5行目ではアリティに2つの自然数をとり，一つの自然数を返す自然数上の加算演算 + を定義している．シンタックスの定義部での“_ **+** _”は“_”の位置にアリティを記述することを表しており，ここでは“**+**”が中置記法によって記述されるということを表現している．

ここまでが指標の説明である．指標によってソート Zero・NzNat・Nat とその上の演算により，項が定義される．各ソートは要素に次のような項を持つ．

Zero　要素として唯一の項0を持つ．

NzNat　s(0), s(s(0)), s(s(s(0))), $\cdots$．0に1回以上 s 演算をかけた全ての項．

Nat　Zero, NzNat が持つ全ての要素．また 0 + 0, s(0) + 0, (0 + 0) + s(s(0)) 等，加算演算で作られる項も全て Nat に含まれる．

指標だけでは，0 と 0 + 0 は異なるものとして区別される．これらが等しいものであるということは等式によって定義される．

次に 7 – 9 行目の等式について説明する．7 行目では等式で使用する変数 P,Q を宣言し，それぞれ Nat というソートを持つと定義されている．8,9 行目で加算に関する等式を定義している．8 行目の等式では，0 に任意の自然数 P を足したものは P に等しいと定義している．9 行目の等式では 1 以上の任意の自然数と 0 以上の任意の自然数に対する等式を定義している．CafeOBJ では等式を左辺から右辺の書換え規則とみなし，項を自動的に簡約することができる．例を用いて簡約の様子を示す．

`0 + 0 = 0`	`s(s(0)) + s(0) = s(s(0) + s(0))`
	`= s(s(0 + s(0)))`
	`= s(s(s(0)))`
		`(s(0) + 0) + s(s(0)) = s(0 + 0) + s(s(0))`
		`= s(0) + s(s(0))`
		`= s(0 + s(s(0)))`
		`= s(s(s(0)))`

図 3.2: 等式による簡約の例

全ての簡約において，モジュール中に定義した 2 つの等式だけで簡約が完了している．CafeOBJ では，定義した仕様を高効率に簡約することが可能であるという特徴を持っており，大規模な仕様に対しても十分な処理能力を持っている．

3.1.1 振舞仕様

振舞仕様 (Behavioural Specification) は CafeOBJ 仕様で状態遷移機械を記述するのに適した仕様である．振舞仕様は通常の状態遷移機械のように，変数の組によって状態を表すのではなく，状態空間全体をブラックボックスで表現する．振舞仕様ではこのブラックボックスを表すために隠蔽ソート (Hidden sort) いう特別なソートを使用する．また隠蔽ソートに対して，通常のソートを可視ソート (Visible sort) と呼ぶ．

振舞仕様による状態遷移機械は隠蔽ソートに含まれる 1 つの状態であるが，どの状態であるかを直接特定することはできない．観測演算 (Observation) によって現在の状態に対する一部の情報を知り，操作演算 (Action) によって状態を変化させる．状態の遷移は，操作演算を行う前後の状態に対する観測結果の変化によって定義される．また初期状態を与えるため，隠蔽ソート上の定数を与え，定数に対する観測結果によって初期状態を定義する．

自然数に対するスタックを例にし，振舞仕様についてより詳しく説明する．スタックは操作演算 push によりスタックに自然数を一つ積み，操作演算 pop により一番上の自然数を取り除く．また観測演算 top により，一番上の自然数を読み出すという仕様である．

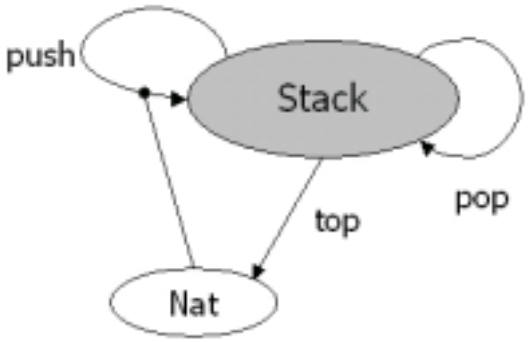

図 3.3: 自然数スタックの振舞仕様に対する ADJ 図

```
 1: mod* STACK{                              --  コメント
 2:  pr(NAT)                                 -- モジュールの輸入
 3:  *[ Stack ]*                             -- 隠蔽ソートの宣言
 4:  bop push   : Stack Nat  -> Stack  -- 操作演算
 5:  bop pop    : Stack      -> Stack  -- 操作演算
 6:  bop top    : Stack      -> Nat    -- 観測演算
 7:  op  empty  :            -> Stack  -- 初期状態
 8:  var  S : Stack
 9:  var  P : Nat
10:  eq  top(push(S,P)) = P .
11:  eq  top(pop(push(S,P))) = top(S) .
12:  eq  top(empty) = 0 .
13: }
```

図 3.4: 自然数スタックの振舞仕様

図3.3は，自然数のスタックの振舞仕様を図示した ADJ 図である．ADJ 図は振舞仕様の隠蔽ソート・可視ソートと操作演算・観測演算の関係を図示したものである．各楕円はソートを表している．色付きの楕円 Stack がスタックの状態空間を表す隠蔽ソートであり，通常の楕円 Nat が自然数を表す可視ソートである．Stack は操作演算 push により引数に取る Nat を Stack に一つ積み上げる操作を行う．また操作演算 pop により，Stack の一番上のものを取り除く操作を行う．各操作演算はある Stack をもらい，それに何らかの操作を行って別の状態の Stack を返す．このため， push・pop は Stack から Stack に戻る矢印で表さる．観測演算 top は， Stack の一番上に乗っている自然数を返すので，Stack から Nat に伸びる矢印で表される．観測演算により，現在の状態の一部の情報について知ることができる．

図3.3を実際に振舞仕様で記述すると図3.4の様になる．STACK モジュールは 2 行目で Nat を定義した NAT モジュールをインポートし，CafeOBJ に既に定義されている NAT モジュール上のソート Nat とその上の演算を再利用することができる．3 行目で隠蔽ソート

Stack を宣言する．“***[**” と “**]***” で囲まれたソート宣言は，隠蔽ソートの宣言となる．2,3 行目によりソート Nat と Stack を使用することができるようになる．

4–6 行目で各演算を宣言する．各演算は次のような構文で定義される．

“**bop**” 演算名 “**:**” アリティ “**–>**” コアリティ

操作演算 push は，Stack と Nat をアリティにとり，コアリティに Stack を返す演算である．操作演算 pop は，Stack をアリティにとり，コアリティに Stack を返す．観測演算 top は，Stack をアリティに取り，コアリティに Nat を返す．操作演算とは，アリティとコアリティに同じ隠蔽ソートを取る演算のことを言う．観測演算は，アリティに隠蔽ソートを取り，コアリティに可視ソートを取る演算である．7 行目で，初期状態を表す定数 empty を宣言する．空のスタックを意味する定数である．ここまでの宣言した，ソートと演算により仕様から記述される項の文法が定義される．ソートと演算をあわせて指標 (signature) と呼び，Σ で表す．

8–9 行目では，等式を定義するのに使用する変数を宣言する．変数 S, P はそれぞれ Stack, Nat 上の変数である．

10–12 行目で等式を定義する．指標が仕様で記述される項の文法を定義するのに対し，等式は仕様の意味を与える．振る舞い仕様では等式により状態遷移機械の遷移関係を定義する．等式は次のような構文を持つ．

“**eq**” 左辺 “**=**” 右辺 “**.**”

もしくは

“**ceq**” 左辺 “**=**” 右辺 “**if**” 条件文 “**.**”

左辺，右辺，条件文は指標と変数により記述される項である．10 行目の等式は，スタックに P を乗せると，一番上には P が乗っていることを意味する．11 行目の等式は 任意の状態 S に対して push して pop した状態の観測 top は，元の状態 S に対する top と等しいことを表している．12 行目では初期状態 empty に対する観測 top は常に 0 を返すことを表している．

以上により，状態遷移機械 Stack の定義は完了である．このスタックは push により自然数を積み上げ続けていくので，無限状態を持つ仕様である．振舞仕様は，任意の状態に対する各観測演算の結果が定義されていなければならない．Stack の仕様では初期状態から初め，push, pop で到達可能な全ての状態に対して top の値を一意に決定できるように定義されている．

普通，状態遷移機械は変数の組で状態 S を表し，遷移関係 R を状態の直積の部分集合 $S \times S \subseteq R$ で与える．そして初期状態 I を状態 S の部分集合 $I \subseteq S$ で与える．これに対し振舞仕様は，状態遷移機械の内部については一切感知せず，振る舞いという外側から見た情報だけで状態遷移機械を定義する．このため振舞仕様は 2 つの状態機械の等価性を調べ

るのに適している．また普通の定義による状態遷移機械よりも，より普遍的に状態遷移機械を定義する．

3.2 SMV

SMV は E. M. Clarke と K. L. McMillan によって設計・実装された検証のための言語である．現在では McMillan によって実装された SMV (他と区別するため Cadence SMV [cad] と呼ぶ) の他に，IRST (Trento, Italy) で実装されておりオープンソースである NuSMV2 [CCGR98, nus] が特に有名である．

3.2.1 SMV の状態遷移機械

SMV 言語は，有限状態機械を定義する言語である．状態を変数の組で表現し，遷移関係を状態と状態の直積上の関係で与える．また SMV によって検証する性質を時相論理 (Temporal Logic) によって与える．

SMV 言語は，状態遷移機械で使用する変数を最初に確定し，動的に増やすことはできない．このため状態の遷移系列に依存せずに，到達する可能性のある状態を限定することができる．図3.5は，2 変数からなる状態遷移機械である．変数 x は x_0, x_1, x_2, x_3 の値をとり，変数 a は a_0, a_1, a_2 の値をとりうる．このため状態空間は，初期状態から到達できる状態は 7 つであるが，変数の定義から到達する可能性のある状態数は $|x| \times |a| = 4 \times 3 = 12$ にであることが分かる．SMV は検証を行うために，変数宣言から到達する可能性のある状態空間が全て列挙できる必要がある．

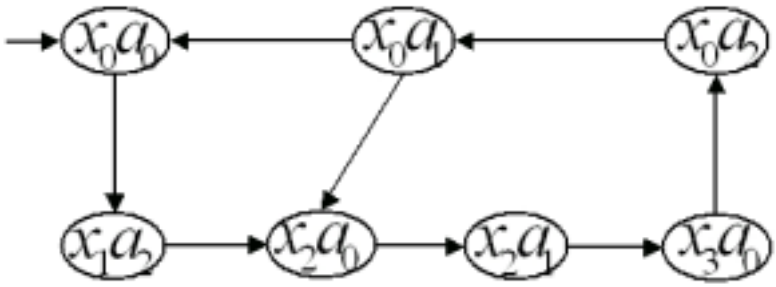

図 3.5: SMV 言語で扱う状態遷移機械の例

```
 1:  MODULE main
 2:  VAR
 3:    x : {x0, x1, x2, x3};
 4:    a : {a0, a1, a2};
 5:  ASSIGN
 6:    init(x) := x0;
 7:    init(a) := a0;
 8:
 9:    next(x) :=
10:    case
11:      x = x0 & a = a0 : x1;
13:      x = x0 & a = a1 : {x0, x2};
14:      x = x1          : x2;
15:      x = x2 & a = a1 : x3;
16:      x = x3          : x0;
17:      1               :  x;
18:   esac;
19:   next(a) :=
20:   case
21:     x = x0 & a = a0 : a2;
22:     a = a1          : a0;
23:     x = x0 & a = a2 : a1;
24:     x = x1          : a0;
25:     x = x2 & a = a0 : a1;
27:     x = x3          : a2;
28:   esac;
29:  SPEC AG !(x = x2 & a = a2)
30:  SPEC AG (a != a2)
```

図 3.6: 図3.5に対する SMV 仕様

図3.5の状態遷移機械を SMV で記述したものが図3.6である．SMV の仕様はいくつかの部分から構成されており，まず **MODULE** でモジュールの宣言を開始する．**main** モジュールは全ての SMV 仕様に必ず一つ含まれており，**main** から参照できる変数の組を SMV の状態とする．

VAR により変数宣言モードになる．x, a があるので，この 2 つの変数からなる状態空間となる．それぞれの変数は “**{**” と “**}**” の間に列挙された値のみをとることができる．

ASSIGN 以降に遷移規則を記述する．**init**(変数名) **:=** 値; で初期状態を，**next**(変数名) **:=** 値; で変数の遷移後の値を定義する．まず，6–7 行より初期状態 $(x, a) = (x_0, a_0)$ に設定される．9–18 行により，変数 x についての遷移規則を記述している．x の遷移規則は case 文により条件分けされている．case 文の中は “条件 : 値; ” という形をしており，条件を満たす時に値を取ることができる．値を集合により与えると，集合の中から一つを非決定的に選択し次の値とする．この場合 13 行目で x0 か x1 が非決定的に選択される．

SPEC により，検証する性質を記述する．検証する性質はCTLに従った時相論理を用いて，初期状態から到達できる状態に関する性質を記述する．29行目の性質は常にx = x2 & a = a2ではないことを表している．この状態は初期状態から到達できないので，性質は満たされる．30行目の性質は，常にa != a1であることを表している．これは初期状態から到達できる状態であるため，SMVによって自動的に反例が報告される．

表 3.1: SMVにより報告される反例

	1	2	3	4
x	$x0$	$x1$	$x2$	$x2$
a	$a0$	$a2$	$a0$	$a1$

縦軸に変数，よく軸に時間を取った状態遷移列が反例として得られる．1の列が初期状態を表し，2, 3, 4と順に遷移していく．4の状態が性質を満たさない状態である．1から4は安全性を満たさない最も単純な遷移列である．この反例より，性質を満たさない原因を調査し，仕様を修正する．

次にSMVで書かれた別の状態機械を図3.7に示す.

```
 1:  MODULE main
 2:  VAR
 3:    request : boolean;
 4:    state   : {ready, busy};
 5:  ASSIGN
 6:    init(request) := TRUE, FALSE;
 7:    next(request) := TRUE, FALSE;
 8:
 9:    init(state) := ready;
10:    next(state) := case
11:                     state = ready & request  : busy;
12:                     state = ready & !request : ready;
13:                     1 : {ready, busy};
14:                   esac;
15:
16:  SPEC AG AF state = ready
```

図 3.7: SMV仕様による応答システム

応答システムを記述したSMV仕様である．二つの変数request，stateがありそれぞれブール型，列挙型である．`request`に対する初期値，遷移規則は共に{`TRUE`, `FALSE`}と

なっている．これは初期値，遷移後の値として可能なものを列挙している．どの値をとるかは決まっておらず，非決定的に選択される．SMV 処理系はこのような場合についても全ての場合を網羅的に探索する．state に対する初期値として ready が設定される．遷移規則は条件付であり，ready & request が真となる時は busy に，それ以外の場合は ready，busy から非決定的に選択される．

検証する性質は，どの状態においてもシステムがいつかは ready 状態になるという性質である．これは表3.2に示すような反例がある．

表 3.2: 応答システムに対する反例

	1	|: 2 :|
request	TRUE	FALSE
state	ready	busy

表から，初期状態で (request,state) = (TRUE, ready), 次の状態で (request,state) = (FALSE, busy) であることが分かる．時間軸の 2 は|:と:|で囲まれている．これは 2 の状態が永遠に続く場合を示している．この時実行している処理がいつまでたっても終わらないので，応答システムはいつまでも ready にはなれず，仕様を満たさない．

第 4 章

証明論的手法とモデル検査的手法を併用した検証法

本章ではまず振舞仕様と振舞仕様より生成した SMV 仕様の性質の違いについて述べる．本研究で実装した仕様変換器 C-TRAS で扱うことのできる仕様について述べ，次に生成される SMV 仕様の状態遷移機械の動作について説明する．そして仕様の変換法について述べる．C-TRAS の実装を説明した後に変換の正当性について述べる．最後に C-TRAS を利用した検証法について述べる．

4.1 振舞仕様と SMV 仕様の比較

振舞仕様も SMV 仕様も，どちらも状態遷移機械を記述し検証するための仕様である．そのため，振舞仕様と SMV 仕様の間には，表4.1のような対応関係をとることができる．

SMV での状態は変数の組で表される．振舞仕様では状態は陽には定義されず，観測演算によって状態の一部について知ることができる．しかし観測演算がアリティに Nat を取る時，無限個の観測演算があるのに等しく単純に等価ということはできない．

振舞仕様では観測演算から無限の状態を観測できるため，無限状態を取りうる仕様である．これに対し SMV では有限の値をとる有限個の変数の組で状態を表現するため，有限状態の仕様である．SMV による検証の大きな利点である全自動の検証・反例の提示を利用するためには，振舞仕様の状態を有限化する必用がある．

無限状態機械を有限状態機械に変換する方法として，抽象化による方法と制限による方法がある．

- 抽象化

  抽象化 (Abstraction) [CGL94] は，状態爆発を緩和するために非常に活発に研究

表 4.1: 振舞仕様と SMV 仕様の比較

	振舞仕様	SMV 仕様
状態	ブラックボックス (観測演算で参照可)	変数の組
遷移規則	等式で記述した 遷移規則	命題論理式による 遷移規則
モデルの種類	無限状態を含む 振舞仕様	有限状態に限定した 状態機械
検証する性質	全状態で P が成り立つ	AG P
検証	半自動	全自動

されている技術である．抽象化は，状態遷移機械が持つ性質を保ったまま状態を削減する技術である．このため，元の仕様に対する検証と抽象化後の仕様に対する検証は，常に同じ結果を返す．

現在の技術では，抽象化法はヒューリスティクスに依存しており，常に適用できることが保証されない．しかし無限状態の仕様を帰納的に証明できるならば，無限仕様に対する抽象化が存在することが示されている． [KP99]

- 制限

  制限は，抽象化と同様に状態空間を削減する方法である．制限による方法は，無限状態機械から状態機械の一部を切り取る方法といえる．当然のことながら，切り捨てられた状態機械に含まれる誤りを発見することはできなくなる．このため制限によって得られた状態機械に対して検査を行った結果，反例が見つからなかったとしても元の無限仕様に誤りが無いことを示すことはできない．しかし反例を得ることができた場合，この反例は元の無限仕様にも必ず含まれることが保証される．

  抽象化による状態の有限化は容易に行えないのに対し，制限による有限化は範囲を決めるだけであり，容易に有限仕様を得ることができる．C-TRAS では制限による方法を実装しており，容易に有限状態機械を得ることができる．ただし，制限によるモデル検査は反例発見が目的であるため，反例を含みそうな状態空間を適切に選ぶ必要がある．

  現在の実装で抽象化を行う場合は，振舞仕様を書き直し，抽象化したモデルを

振舞仕様を作成する必要がある．

4.1.1 制限情報の与え方

振舞仕様に対して，状態空間を有限化するために制限情報を与える．観測演算・操作演算に現れる Nat, NzNat は状態空間が無限であるので，Nat, NzNat に対して制限情報を与える．図4.1に示す例で制限情報の書き方について説明する．

制限情報は “** ” で始まるコメント行で与える．観測・操作演算を定義した直後に “** ” コメントが現れ，かつコメント中の最初の単語が limit である時，　アリティの制限情報 “->” コアリティの制限情報　という構文で制限情報を仕様から読み込む．アリティ，コアリティの位置に現れる “*” は任意という意味で，制限情報をユーザから与えないことを意味している．隠蔽ソートに対しては与えることができず，Bool に関しては初めから有限なので与える必要は無い．

```
1:  *[ Sys ]*
2:  bop obs1 : Sys Nat   -> Nat . ** limit * 4 -> 5
3:  bop obs2 : Sys Bool  -> Nat . ** limit * * -> 3
4:  bop act1 : Sys Bool  -> Bool .
5:  bop act1 : Sys Bool  -> Sys .
6:  bop act2 : Sys NzNat -> Sys . ** limit * 3 -> *
```

図 4.1: 制限情報の与え方

制限情報は，ソート上の要素をいくつにするかを指定する．Nat に対して n と指定すると，0 .. (n-1) という型に変換され， NzNat に対して m と指定すると 1 .. (m-1) という型に変換される．Nat や NzNat を含まない操作・観測演算に対しては制限情報を記述する必要は無い．逆に Nat や NzNat を含む操作・観測演算に対して指定しない時は C-TRAS によってエラーが報告される．

4.1.2 生成する状態遷移機械

振舞仕様では操作演算適用後の観測結果を求めるために，簡約を繰り返し行う．簡約は，ある状態を表現した項をそれと等しくより簡潔な表現の項へ変換する．

SMV でも同様に 1 回の簡約動作を 1 回の遷移として定義することは可能である．しかしこの方法は SMV 上で扱う状態数を不必要に大きくするため効率的ではない．C-TRAS によって生成される SMV 仕様は，1 回の状態遷移で 1 回の操作演算を適用する．1 回の操作で得られる結果について，CafeOBJ と SMV で大きく異なるが，両者の状態遷移機械の等価性を保つように遷移機械を生成する．

制限により有限状態遷移機械になった振舞仕様に対して，C-TRAS は観測結果の組を状態とみなす．図4.2に示す状態遷移機械で，各 x_0, x_1, x_2, x_3 は観測値を表し，各 a_0, a_1, a_2 は操作演算名とする．初期状態 x_0 で操作演算 a_0 を行うと x_1 に遷移し，a_1 を行うと x_2 に，a_2 を行うと x_3 に遷移する．

これに対し，有限化された振舞仕様を SMV で表現したものを図4.3に示す．SMV の状態遷移機械には遷移にラベルをつけることができない．振舞仕様を SMV で状態を表現するには，観測演算の結果だけではなく，次に実行する操作演算に関する情報も状態を表す変数に追加する．よって x_0a_0 からは x_2a_0 や x_3a_0 に遷移することはできない．このようなモデル化では状態数が増えるという問題点がある．しかし検査から反例を得た時，SMV が示す結果から容易に振舞仕様での反例に変換できる．また制限された振舞仕様と等しい振る舞いをする状態機械を容易に定義できるという利点がある．

ある状態で複数の遷移が可能な SMV 仕様では遷移先は SMV によって網羅的に選択される．図4.2の振舞仕様では，初期状態から 1 回の遷移で到達する状態数が 4 状態である．これに対し図4.3の SMV で表現したモデルでは 2 倍近いの 7 状態である．観測演算数が増えたり，引数が増えるとより大きくなっていく．これは検証時にオーバーヘッドとして現れてくる．しかし，SMV は初期状態から順番に経路をたどるわけではない．検証する性質の各部分項を BDD によって表現し，集合上の演算で網羅的な検証を行う．

4.2 仕様の変換

本節では，仕様の変換方法について述べる．まず現在の実装で，扱うことのできる振舞仕様について説明する．次に変換した結果，どのような動作をする状態遷移機械が得られるのかについて説明する．次に振舞仕様の観測演算，操作演算・等式から・状態 S・遷移関係 R・初期状態 I を取り出し，SMV 仕様に変換する方法について述べる．

4.2.1 変換に対する制約

本研究で実装した変換器 C-TRAS は，現段階ではいくつかの制約を持っている．

- 可視ソートに対する制約　可視ソートとして使用できるソートは，Nat と Bool に限定される．

  この制約は，仕様を記述する上では大きな制約を課する．CafeOBJ 言語の特徴である，モジュール構造で仕様を記述するなどの有益な機能がこの制約によって使用できなくなる．しかし，可視ソートによる任意のデータ構造は自然数である Nat 上で記述することができ，記述できる問題を制約するわけではない．

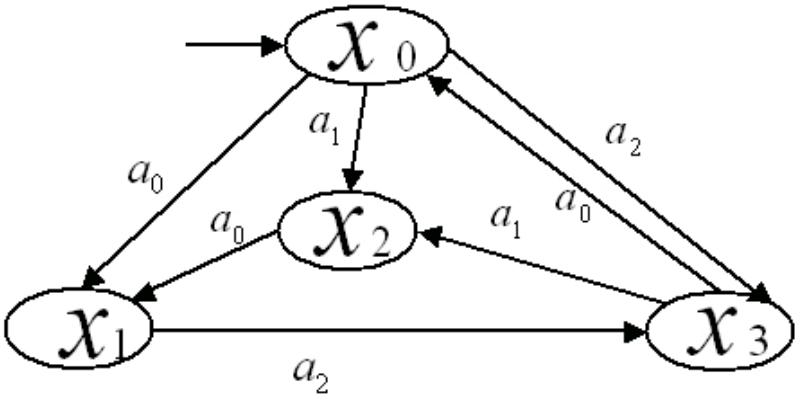

図 4.2: 有限化した振舞仕様の状態遷移

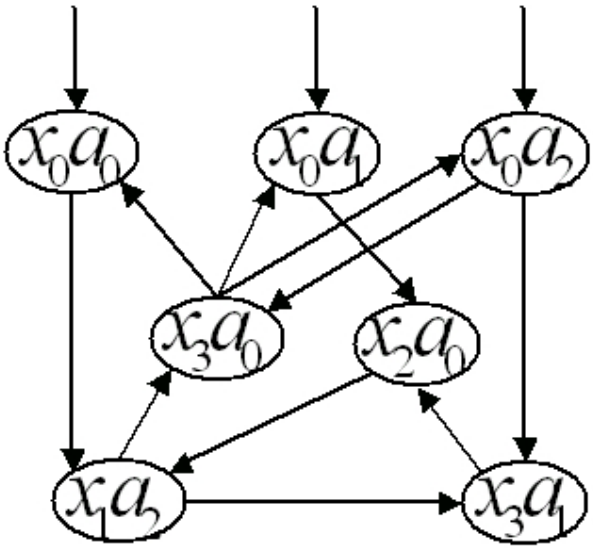

図 4.3: 有限化した振舞仕様に対する SMV の状態遷移機械

- 遷移規則を記述した等式に対する制約　遷移規則を記述した等式の左辺に対して，操作演算・観測演算のアリティに現れる可視ソートを変数のみに制約する．

  この制約は遷移規則として，以下の形をした左辺を持つ等式に限定することを表している．

```
obs(act(S,v1,v2,...),u1,u2,...)
 ただし
   obs        : 任意の観測演算
   act        : 任意の操作演算
   S          : 隠蔽ソート上の変数
   v1,v2      : 可視ソート上の変数
   u1,u2      : 可視ソート上の変数
     (ただしアリティ中のソート順は任意)
```

図 4.4: 遷移規則を記述した等式の左辺に関する制限

この制約を与えることにより，状態遷移機械の任意の 2 状態に対して，全ての観測演算の観測結果が等しい場合，この 2 状態は等しいということができる．この制約は，記述できる仕様のスタイルを制約するが，記述できる問題を制約するわけではない．

また等式の左辺では，操作演算や観測演算のアリティに現れる可視ソートは変数に限定している．この条件を満たさない全ての等式は，次のように変換することが可能である．

変換前

```
var N : Nat
eq obs1(act1(S,1)) = 1 .
eq obs1(act2(S,s N)) = N .
```

変換後

```
var M : Nat
ceq obs1(act1(S,M)) = 1     if M == 1 .
ceq obs1(act2(S,M)) = M - 1 if M - 1 >= 0 .
```

定数に対しては適当な変数で置き換え，変数名 “==” 定数 を条件に追加することで容易に変換できる．演算の中に変数を持つものに対しては，逆演算を使用する．逆演算をかけた結果が変数がとりうる範囲内であるかどうかを条件式に加える．BOOL・NAT 上の演算は，rem のように一意に決まらないものがあるが，有限なので可能な値を列挙することができる．またこの制約は等式の右辺，if 節では適用されない．

- 初期値を記述した等式に対する制約　初期値規則を記述した等式に対して，左辺に現れる隠蔽ソート上の定数の個数を 1 つに制約する．また一つの観測演算のアリティ

に現れる隠蔽ソートを 1 つに制約する．

この制約は初期値の定義として，以下の形をした左辺を持つ等式に限定することを表している．

```
obs(ini,u1,u2,...)
 ただし
   obs        : 任意の観測演算
   ini        : モジュール中で宣言された隠蔽ソート上の定数
   u1,u2      : 可視ソート上の変数
     (ただしアリティ中のソート順は任意)
```

図 4.5: 遷移規則を記述した等式の左辺に関する制限

- 遷移規則を記述した等式の右辺・条件式に対する制約　操作演算のアリティに制限を行うことにより，演算の種類が有限になり，観測演算のアリティに制限を行うことにより，振舞仕様を有限個の変数の組として表現できるようになる．制限により残した変数の遷移規則は，制限により無視する変数の値に依存してはならない．

  これは制限した状態遷移機械が，元の振舞仕様で到達不能な状態へ到達しないようにするために必要である．変換した結果，制限により無視される変数が遷移規則に現れる場合，SMV によってエラーとして報告される．

最初の制約により，任意の可視ソートを使用することはできない．実際には，これは非常に強い制約である．しかし任意の可視ソートを扱いながら，仕様の等価性と実行効率を両立する仕様を生成することは容易ではない．

任意の可視ソートを使用しても元の仕様と同じ意味を保証する変換法として，SMV の上で項書換えを実行する方法が考えられる．CafeOBJ の 1 回の項書換えと SMV 上の 1 回の遷移を対応付けるように変換する．実際の適用事例として NSPK プロトコルに対し，この方法に基づき手動で振舞仕様から SMV 仕様になるべく機械的に変換した．NSPK プロトコルは欠陥のあるプロトコルであり，SMV により反例が得られる予定であった．2Gbyte のメモリを与えてモデル検査を行ったが，状態爆発により検証を最後まで行う事はできなかった．このため現状では，効率よく解くことができるよう可視ソートに制約を与えている．

仕様変換器 C-TRAS によって変換が可能な振舞仕様の構文は，図4.6のように与えられる．構文 A B は，A の後に B が続く構文を表し，A | B は，A か B の構文の選択を表す．A* は 0 回以上 A が連続して現れる構文を表し， A+ は 1 回以上 A が連続して現れる構文を表す．また A [B] は A の後に，B があっても無くても良い構文を表す．{ と } で囲まれた構文はそれで一つの構文として扱われ，直後に現れる + や * を括弧全体に適用する．

表 4.2: 入力言語の予約語の省略形

正規形	省略形
associativity	assoc
module*	mod*
protecting	pr
ceq	cq

また，各構文の間には一つ以上の*whitespace*で区切られる．

また予約語の中には，表4.2のように省略形を持つものがある．

4.2.2 SMV 仕様での状態

振舞仕様では隠蔽ソートによって表されるブラックボックス化された状態に対して，任意の観測演算に任意の引数を与えて状態を変化させていく．これに対応する SMV 仕様を作るため，1) 観測演算によって観測される全ての観測結果の組を現在の状態として定義する．これにより状態機械の各状態を SMV で表現できる．しかしこれだけの情報では，どの操作演算によって遷移するかという情報が得られない．このため，2) 状態に操作演算を実行するために必用な操作演算名とその引数，を追加する．SMV 仕様での状態は，1, 2 で得られる変数の組である．

観測演算に関する SMV 上の変数

観測演算の返す値の組は，表4.1の対応関係で示したように振舞仕様における一つの状態を表す．観測演算を SMV 仕様の変数に変換するために，アリティに可視ソートを持つ観測演算と，持たない観測演算に分けて考える．

- アリティに可視ソートを持たない観測演算 は表4.3が示すように変換する．

  以下に変換例を示す．

  振舞仕様の観測演算

```
bop turn   : Sys -> Bool .
bop state  : Sys -> Nat . ** limit * -> 5
```

  対応する SMV 仕様

```
VAR
  turn  : boolean;
  state : 0 .. 4;
```

  アリティに可視ソートを取らない観測演算は，観測演算名を変数名とし，変数の型をコアリティに対応する SMV 型とすることで変換できる．コアリティに

module ::= “**module***” *module_name* “**{**” *module_body** “**}**”
module_name ::= *identity*
module_body ::= *hsort_def* | *import* | *operation* | *var_def* | *axiom*

hsort_def ::= “***[**” *hsort_name* “**]***”
hsort_name ::= *identity*
sort_name ::= *hsort_name* | *datatype*

import ::= “**protecting**” “(” *import_modules* “**)**”
import_modules ::= *import_module* [“**+**” *import_modules*]
import_module ::= “**NAT**” | “**BOOL**” | “**SAFETY**”

operation ::= *single_operaton* | *multi_operation*
single_operaton ::= {“**op**” | “**bop**”} *op_name* “**:**” *sort_name** “**–>**” *sort_name* [*attribute_list*] [“**.**”] [*limit_def*]
multi_operation ::= {“**ops**” | “**bops**”} *op_names+* “**:**” *sort_name** “**–>**” *sort_name* [*attribute_list*] [“**.**”] [*limit_def*]
op_name ::= *std_op_name* | *mixfix_op_name*
op_names ::= *std_op_name* | “(” *op_name* “**)**”
std_op_name ::= *identity*
mixfix_op_name ::= {*identity* | “**_**”}+
attribute_list ::= “**{**” *attributes+* “**}**”
attributes ::= “**l_assoc**” | “**r_assoc**” | “**associativity**” | *prec*
prec ::= “**prec:**” *number*
limit_def※ ::= “**”** “**limit**” *limit_symbol** “**–>**” *limit_symbol*
limit_symbol ::= *number* | “**”

var_def ::= *single_var_def* | *multi_var_def*
single_var_def ::= “**var**” *var_name* “**:**” *datatype*
multi_var_def ::= “**vars**” *var_name+* “**:**” *datatype*
var_name ::= *identity*
datatype ::= “**Nat**” | “**NzNat**” | “**Zero**” | “**Bool**”

axiom ::= *equation* | *cequation*
equation ::= “**eq**” *leftterm* “**=**” *rightterm* “**.**”
cequation ::= “**ceq**” *leftterm* “**=**” *rightterm* “**if**” *ifterm* “**.**”

identity ::= *self_terminating_code*, “**.**”(ピリオド), “**””**”(ダブルクォート) 以外の表示可能な ASCII 文字からなる文字列
self_terminating_code ::= “(” | “)” | “{” | “}” | “[” | “]” | “,”
number ::= 0 から 9 の文字からなる文字列
leftterm ::= 図4.4または図4.5によって制限される*term*.
rightterm ::= 操作演算を使用しない任意の*term*
ifterm ::= 操作演算を使用しない任意の *term* で，Bool のソートを持つ*term*
term ::= 輸入されたモジュールが持つ任意の*operation*と，モジュール中で定義された任意の*operation*から作成される．
whitespace ::= 空白文字 (␣)，タブ文字，改行記号，または*comment* のいずれか
comment ::= “**” または “**--**” から行末まで

※ コメント中に記述するため，1 行に収める必要がある．

図 4.6: 仕様変換器 C-TRAS の入力となる振舞仕様の構文

表 4.3: 振舞仕様のソートと SMV の型の対応関係

振舞仕様のソート	SMV 仕様での型
Bool	boolean
Nat	制限情報より個数(n 個) を読み, 0 .. (n-1)
NzNat	制限情報より個数(n 個) を読み, 1 .. (n-1)
Zero	0 .. 0

Bool を持つ turn は boolean 型を持つ変数として変換される．コアリティに Nat を持つ state は，制限情報により，この例では 0..4 に変換される．ここで Sys は隠蔽ソートとして定義されているものする．

- アリティに可視ソートを持つ観測演算 の場合，アリティにとる値によって観測結果がそれぞれ異なることになる．このため，アリティにとりうる値の個数だけ変数を宣言する必要がある．ここでは配列を用いて必用な数変の個数を用意する．

  振舞仕様の観測演算

```
bop flag   : Sys Bool     -> Bool .
bop state  : Sys Nat      -> Nat .  ** limit * 3 -> 4
bop pstate : Sys Nat Nat -> Bool .  ** limit * 2 3 -> *
```

  対応する SMV 仕様

```
VAR
  flag   : array 0 .. 1 of boolean;
  state  : array 0 .. 2 of 0 .. 3;
  pstate : array 0 .. 1 of array 0 .. 2 of boolean;
```

  flag はアリティに Bool をとる．Bool は true と false の 2 個のみであることが分かっているので，flag は 0 .. 1 の範囲をとる boolean 型の配列として定義される．state はアリティに Nat をとる．Nat の場合は制限情報より個数を求める．ここでは 3 個と指定されているので 0 .. 2 の範囲をとる配列として定義される．pstate のように複数の可視ソートを持つ場合，SMV 仕様では多次元配列として定義される．このようにアリティに可視ソートをとる場合は，事前にそのソートがとりうる値の個数 n を求め，0 .. (n-1) の要素を持つ配列として定義する．

以上により，アリティに任意個の Nat, Bool をとる任意の観測演算に対して，SMV 仕様上の変数に変換することができる．次に操作演算を実行するために必用な変数について説明する．

操作演算に関する SMV 上の変数

各状態遷移で任意の操作演算を行うために，操作演算名とその引数に相当する変数を SMV 仕様に追加する．例えば，操作演算として以下に示すものがあるとする．

```
*[ Sys ]*
bop act1 : Sys Bool Bool Nat -> Sys . ** limit * * * 3 -> *
bop act2 : Sys Bool NzNat    -> Sys . ** limit * * 2   -> *
bop act3 : Sys Nat Nat       -> Sys . ** limit * 3 3   -> *
```

このとき，操作演算に対応して作られる変数は以下の様になる．

```
-- 操作演算名に対応する変数
  __action : {act1, act2, act3};
-- 操作演算の引数に対応する変数
  _bool0 : boolean;
  _bool1 : boolean;
  _nat0  : 0 .. 2;
  _nat1  : 0 .. 2;
```

__action は全操作演算名の集合からなる変数であり，この値により次に実行する操作演算が決定する．

_bool0, _bool1, _nat0, _nat1 は操作演算の引数に対応する変数である．一度に実行できる操作演算は一つであるので，同時に必用になる最小限の変数にまとめる．Bool に関しては act1 で同時に 2 つ使い，3 個に制限された Nat は act3 によて同時に 2 個使用する．このため，必要最小限の変数は上で示したように 4 つとなる．act2 では，Nat のサブソートである NzNat が使用されている．Nat が 0 以上の整数を表すのに対し，NzNat は 1 以上の整数を表す．NzNat に対する制限情報は 2 となっているので，1 から 2 要素，つまり 1 と 2 を値としてとりうることになる．サブソート関係にある変数は必要に応じてスーパーソート上の変数を利用して余分な状態を削減する．この場合，サブソートをスーパーソートで代用する処理により，状態数は半分に抑えられる．これら処理を行う事で，むやみに状態数を増やすことを抑制する．

全ての観測演算に対して変換を行い，全ての操作演算から __action 変数と，引数に必用な変数を決定する．これらの操作により，SMV 仕様で必要になる全変数が得られる．

4.2.3 等式に対する変換

以下では等式を変換する手続きを定義するため，まず等式を表す項と SMV の式の内部表現について説明する．次に項を SMV 上の式に変換方法について述べる．項の変換法を使用して複数の等式から SMV での遷移規則を作成する方法について述べる．

項の表現

処理系では，CafeOBJ が扱う項を木で表現して使用している．

木の各ノードは **op**, **bop** によって宣言された演算，または **var** によって宣言された変数である．木の葉は true, false, 0 などの定数を表す演算と変数から構成され，木の枝はアリティに 1 つ以上のソートを取る演算によって構成される．

SMV で扱う式も木として表現する．SMV でも自然巣やブールを扱うための基本的な演算があり，CafeOBJ で表すのと同様に SMV 上の演算からなる木として構成する．各ノードの演算には優先順位に関する情報を持たせる．木から文字列に変換する時に，必要に応じて括弧を付加する．

等式は 左辺，右辺，if節の 3 項の組によって表現される．等式`ceq turn(setturn(S,P)) = P if P < 2 and state(S,P) == 0 .`(≡ E とする) を木で表現すると，図4.7，図4.8，図4.9のようになる．

これらの木に対して，変換や必要な情報の抽出を繰り返すことで SMV 仕様を得ていく．

項の変換

等式の右辺や if 文で現れる項は，変数と Nat · Bool 上の演算，観測演算によって構成される．それぞれの場合に対する変換法を説明する．この変換法により，C-TRAS が扱うことができる仕様上の任意の右辺・if 文の項に対して，それと等価な SMV 上の式を得ることができる．

変数 については，名前替えを行い SMV 上の変数ノードへと変換する．

- 変数が操作演算の引数に現れる変数である場合，次の規則に従って変換される．

  “_” *sort_name number* を連結した文字列.
  ただし，*sort_name*, *number* は図4.6によって定義される文字列.

  数字は同じソートを持つ変数を区別するために 0 からつけられる．例えば `_nat0`，`_bool0`，`_bool1` のように変数名が作られる．

- 変数が観測演算の引数に表れる変数である場合，次の規則に従って変換される．

  “**_var**” *number* を連結した文字列.
  ただし，*number* は図4.6によって定義される文字列.

  数字は，観測演算のアリティの位置より決定される．隠蔽ソートを無視し，左の可視ソートから順に `_var0`，`_var1`，`_var2`，... という名前がつけられる．

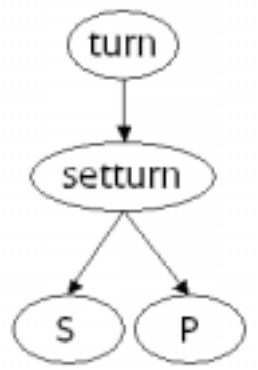

図 4.7: 等式 E の左辺の内部表現

図 4.8: 等式 E の右辺の内部表現

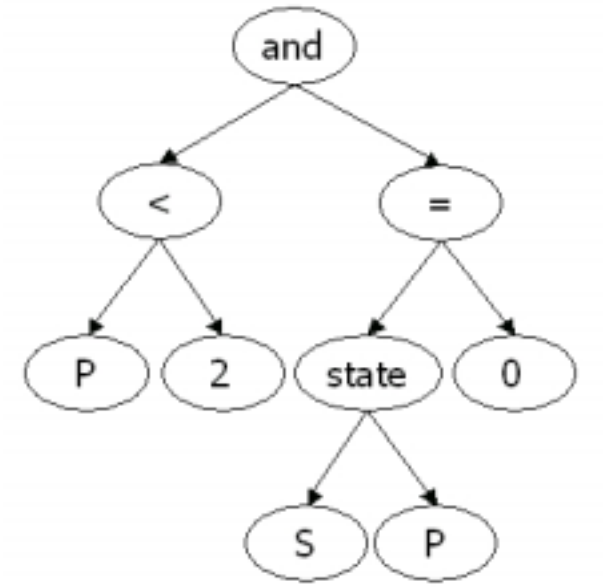

図 4.9: 等式 E の if 節の内部表現

CafeOBJ 仕様ではパターンマッチが可能であるため，同じ変数が左辺で繰り返し現れる可能性がある．操作演算と観測演算の引数に同じ変数が現れる場合，観測演算の変換を優先し "**_var**" *number* で名前が付けられる．操作演算のアリティ中に同じ変数が現れる場合，同じ変数名に変換する．

Nat 上 · Bool 上の全ての演算 については，SMV 上の演算と明らかな対応関係をとることができる．このため，Nat と Bool の演算に関しては事前に対応関係を表4.4(P.38)に示すように作成することで変換が可能である．この表に従い，各ノードを再帰的に変換する．

観測演算 は，観測結果を SMV 上で変数として扱う．アリティが隠蔽ソートだけである観測演算は，対応する SMV 上の変数ノードに変換する．アリティに Bool や Nat が現れる場合，観測演算に対応する変数ノードは配列である．この場合の変換は，1) 各アリティの項を SMV 上の木に変換する．2) 1 で得た木を順番に配列のインデックスとする．

以上の操作で任意の等式の右辺・if 文の項を SMV の式に変換することができる．図4.9の if 節に対する項の変換を例に変換過程を次に示す．

toSMV((P < 2) and (state(S,P) == 0))
toSMV((P < 2) & *toSMV*(state(S,P) == 0)
(*toSMV*(P) < *toSMV*(2)) & *toSMV*(state(S,P) == 0)
(_nat0 < *toSMV*(2)) & *toSMV*(state(S,P) == 0)
(_nat0 < 2) & *toSMV*(state(S,P) == 0)
(_nat0 < 2) & (*toSMV*(state(S,P)) = *toSMV*(0)))
(_nat0 < 2) & (state[*toSMV*(P)] = *toSMV*(0))
(_nat0 < 2) & (state[_nat0] = *toSMV*(0))
(_nat0 < 2) & (state[_nat0] = 0)

ただし *toSMV* **:** 振舞仕様での項 -> SMV の式

各演算は，and が & に，==が=に変換されるなど，表4.4に従い変換される．変数 P は，図4.7で操作演算の引数として現れている．このため，_nat0 へと変換される．観測演算はアリティに P を取るため，state[_nat0] に変換されている．

等式から遷移規則の生成

振舞仕様を SMV で表現するために必用となる変数は，観測演算に対応する変数と操作演算に関する変数がある．それぞれの場合について遷移規則を求める．

- 観測演算に対応する変数の遷移規則

表 4.4: Bool, Nat 上の演算の対応関係

CafeOBJ 上の演算	SMV 上の演算
true	TRUE
false	FALSE
not A	! A
A *and* B	A & B
A *or* B	A | B
A *xor* B	A *xor* B
A *iff* B	A <–> B
A *implies* B	A –> B
A == B	A = B
A =/= B	A != B
0	0
P + Q	P + Q
P − Q	P − Q
P ∗ Q	P ∗ Q
P *quo* R	P / R
P *rem* R	P *mod* R
R *divides* P	P *mod* R = 0
P <= Q	P <= Q
P < Q	P < Q
P >= Q	P >= Q
P > Q	P > Q
P == Q	P = Q
P =/= Q	P != Q
s P	P + 1

※ P, Q は Nat 上の項を表す変数.
R は NzNat 上の項を表す変数.
A, B は Bool 上の項を表す変数.

項の変換で定義した手続きを使用して，一つの等式を対応する遷移規則へと変換する方法について説明する．

1. 遷移規則に関する等式を，観測演算ごとに振り分ける．振り分けたグループごとに 2 以下の処理を行う．
2. 等式左辺の操作演算名を取り出し，__action = 演算名 という SMV の式を作成する．
3. 等式左辺の変数のソートを検査する．変数のソートが，NzNat, Zero である場合，Nat を使用するために，パターンマッチの条件式を生成する．パターンマッチの条件式は変数 P が NzNat の場合，P >= 1 であり，Zero の場合, P == 1 である．変数が Nat, Bool の時は TRUE が条件式となる．作成した条件式は 2 で作成した条件式と and 演算 で結ぶ．
4. 条件付等式であれば，if 節を項の変換手続きを用いて SMV 式に変換し，3 の結果と and 演算で結ぶ．
5. 等式の右辺を項の変換手続きを用いて SMV 式に変換する．
6. 4 で得られた結果と，5 で得られた結果を組にしてリストに追加．グループ内の残りの等式に対して，2 以下を行う．全て行った場合に次へ進む．
7. < 遷移条件，遷移後の値 >の組を要素に持つリストを元に case 文による遷移規則を生成する．

以上の手続きで，図4.10の遷移規則が得られる．

```
next(観測演算に対応する変数名) :=
case
  __action = 等式 1 の action & オーダーソートの条件 & if 節の条件 : 遷移後の値 1;
  __action = 等式 2 の action & オーダーソートの条件 & if 節の条件 : 遷移後の値 2;
  __action = 等式 3 の action & オーダーソートの条件 & if 節の条件 : 遷移後の値 3;
    ⋮                    ⋮                ⋮            ⋮
  __action = 等式 n の action & オーダーソートの条件 & if 節の条件 : 遷移後の値 n;
esac;
```

図 4.10: 等式から変換した SMV の遷移規則の一般形

項の変換規則により，観測演算の変数はアリティの位置より決められる “**_var**” number という名前が付けられている．遷移規則は配列の各要素ごとに全て記述し

なければならない．各遷移規則を記述する時に，この変数は配列のインデックス値に置き換えられる．“**_var**” number に具体的な値を入れることで，それぞれの変数に特化した遷移規則が得られる．

次の観測演算を例に考える．

```
bop obs1 : Sys Nat -> Nat  ** limit * 3 -> 5
```

また観測演算 obs1 に対する遷移規則に R_{obs1} という名前をつける．R_{obs1} は，アリティに Nat をとるため，_var0 という変数を持つ遷移規則になっている．

```
obs1 : array 0 .. 2 of 0 .. 4;
```

に対する遷移規則は，R_{obs1} を変数に_var0 を持つ 1 変数関数とみなす事で，次のように得られる．

```
next(obs1[0]) := R_obs1(0);
next(obs1[1]) := R_obs1(1);
next(obs1[2]) := R_obs1(2);
```

観測演算のアリティに n 個の可視ソートを取るならば，R_{obs1} を n 変数関数とみなし，全ての組合せに対して遷移規則を生成する．

- 操作演算に関する変数の遷移規則

  操作演算から生成される変数には，操作演算名の集合からなる変数 __action と，操作演算を実行する際に必要になる引数に対応する変数がある．操作演算の引数は，そのソート上の任意の項を取ることができる．このため，各変数は常に可能な全ての値を選択できるようにする．これは SMV の非決定的な遷移規則を利用して表現する．

```
init(__action) := {act1, act2, act3};  -- 初期状態の宣言
next(__action) := {act1, act2, act3};  -- 遷移規則の宣言
init(_bool0)   := {TRUE, FALSE};       -- 初期状態の宣言
next(_bool0)   := {TRUE, FALSE};       -- 遷移規則の宣言
init(_nat0)    := {0, 1, 2};           -- 初期状態の宣言
next(_nat0)    := {0, 1, 2};           -- 遷移規則の宣言
```

  遷移する値を { と } で囲んで与えることにより，SMV は集合の中から次の状態を非決定的に決める．可能な全ての組合せの演算・引数で状態遷移を行うため，全遷移系列を検査することができる．観測演算に関する変数の値は，次に実行する操作演算を表すため，__aciton 等の初期状態も非決定的に与える必要がある．

以上の手続きにより，振舞仕様の遷移規則である等式から遷移関係 R を取り出し，SMV の遷移規則を生成する．SMV では各変数の遷移規則はそれぞれ独立に記述し独立に遷移規則が適用される．しかし操作演算名とその引数に対する条件式により，各変数で共通の

操作演算・引数を適用することが保証される．また生成される規則は，各等式と 1 対 1 で対応付けられている．等式が全ての場合について網羅的に記述されていれば，ここで得られる遷移規則は全ての場合について網羅されている．振舞仕様が不完全で全ての場合を網羅していない遷移規則の場合，SMV に夜検証を行うとエラーを報告し停止する．

等式の変換に関して，現在の実装では不十分な点がある．振舞仕様から SMV 仕様に変換するために，状態空間を有限化した．有限化した範囲内の状態から 1 回遷移することにより，範囲外の状態に遷移する可能性がある．現状ではこれらについて何ら対策を施していない．

この問題の回避策として 2 種類考えられる．

1. 範囲外に遷移しないように条件式を加え，常に範囲内で遷移を繰り返す．

   この方法の利点は，ほとんど仕様の修正を加えることなく実現できる点である．また，変換後の仕様も単純である．次の状態への遷移があるにもかかわらず，遷移できないことため誤った結果を報告する可能性がある．ただし安全性の検証ではこの問題は発生しない．

2. over : boolean という変数を加え，一つの変数でも範囲外に遷移したときは over を TRUE にする．

   現在の状態が指定した範囲内か範囲外かを常に over という変数で表す．over が TRUE でない時のみ，通常の遷移を行う．デッドロックが発生しないこと (AG EX TRUE) など重要な性質を調べる上で，前者と比べ正確な結果を返す．状態遷移の変換・安全性の変換が複雑になることと，仕様自体も複雑になるという問題点がある．

範囲外に遷移するような仕様に対し，NuSMV では範囲外に遷移するパスを発見した時点で誤りを報告し停止する．Cadence SMV では個々に遷移規則を評価し，範囲外に遷移する場合はその変数を最小値にセットする．Cadence SMV の場合，本来到達不能の場合であっても到達することが可能になってしまう．Cadence SMV を使用する場合は，この点に気をつけて検証しなければならない．

4.2.4 初期値に対する変換

初期値は隠蔽ソート上の定数と，この定数に対する観測結果を定義する等式により与えられる．初期値に対する変換は，操作演算に対する処理が無い遷移規則の変換とほぼ等しい．異なる点は，遷移規則ならば全ての場合について遷移結果が与えられなければならないが，全ての変数に対し初期値が与えられなくとも良い．初期状態は複数とることができ，初期値の場合，指定しない変数に対する初期値は任意の値である．

1. 初期値に関する等式を，観測演算ごとに振り分ける．振り分けたグループごとに 2 以下の処理を行う．
2. 等式左辺の変数のソートを検査する．変数のソートが，NzNat, Zero である場合，Nat を使用するために，パターンマッチの条件式を生成する．パターンマッチの条件式は変数 P が NzNat の場合，P >= 1 であり，Zero の場合, P == 1 である．変数が Nat, Bool の時は TRUE が条件式となる．作成した条件式は 2 で作成した条件式と and 演算 で結ぶ.
3. 条件付等式であれば，if 節を項の変換手続きを用いて SMV 式に変換し，2 の結果と and 演算で結ぶ.
4. 等式の右辺を項の変換手続きを用いて SMV 式に変換する．
5. 3 で得られた結果と，4 で得られた結果を組にしてリストに追加．グループ内の残りの等式に対して，2 以下を行う．全て行った場合は次に進む.
6. 初期値が記述されていない変数は任意の値を初期値に持つことになる．このため，デフォルト値として変数の定義域の全て要素からなる集合を与え，任意のものが選ばれるようにする．そのためにリストに < 1, { 観測演算が取る全ての値の列挙 } >を追加する．
7. <遷移条件, 遷移後の値 >からなるリストを元に case 文による遷移規則を生成する．

以上の手続きで，次ような初期値を決定する規則が得られる．

```
init(観測演算に対応する変数名) :=
case
  オーダーソートの条件 & if 節の条件 : 初期値 1;
  オーダーソートの条件 & if 節の条件 : 初期値 2;
  オーダーソートの条件 & if 節の条件 : 初期値 3;
        ⋮                ⋮              ⋮
  オーダーソートの条件 & if 節の条件 : 初期値 n;
  1                                : { 観測演算が取る全ての値の列挙 }
esac;
```

図 4.11: 等式から変換した SMV の初期値の一般形

また観測演算がアリティに可視ソートを取る場合，遷移規則と同様に各変数に特化した初期状態を同様の方法により求める．

以上の処理で，仕様と制限情報を与えることにより対応する SMV の状態遷移機械が自動的に得られる．

4.2.5 安全性に対する変換

安全性を記述するために CafeOBJ のモジュールを 1 つ定義する．

```
mod! SAFETY{
  pr(BOOL)
  op safety _ : Bool -> Bool .
}
```

図 4.12: SAFETY モジュール

安全性は，safety から始まる左辺を持つ等式で記述される．

```
eq  safety A = B .
ceq safety A = B if C .
```
ただし A, B, C はそれぞれ操作演算を含まない任意の Bool 上の項

これらはそれぞれ次の様に変換される．

```
AG (A' = B')
AG (C' -> (A' = B'))
```
ただし A', B', C' はそれぞれ A, B, C に対応する SMV 上の式

仕様に変数が現れた場合，それは複数の安全性を定義していることとなる．変数に対して，可能な全ての組合せを網羅的に生成する．

安全性に関する変換手続きをまとめる．

1. 等式の左辺が safety から始まる等式を集める．各等式に対して 2 以降を適用する．
2. 等式の左辺の項，右辺の項，if 節の項をそれぞれ対応する SMV 式に変換する．
3. 等式に現れる変数の名前と制限情報の組を作成する．
4. AG(左辺の項 = 右辺の項) もしくは AG (if 節の項 -> (左辺の項 = 右辺の項)) という SMV 式を作成する．
5. 4 で作成した SMV 式を 3 で求めた制限情報に基づき，全てのパターンを生成する．

以上により安全性が生成され，状態遷移機械の変換と検証する性質の変換が終了する．

4.3 変換の正当性

本研究で実装した仕様変換器C-TRASの変換処理の正当性として，SMVによって報告された反例が元の振舞仕様でも反例であることを示す．

入力となる振舞仕様は，図4.6の構文に従い，4.2.1項で挙げた制約を満たす仕様とする．ここで証明を簡単にするため，操作演算・観測演算のアリティは可視ソートを取らないという条件を加える．これは制限情報により引数の取れる範囲を制限した振舞仕様と等価である．

- 操作演算は引数を取ることにより無限個の操作演算が可能であったが，制限により有限個の操作演算のみを使用する．操作演算の種類を限定した振舞仕様で反例が発見されれば，その反例は明らかに操作演算の種類を限定しない振舞仕様でも反例である．
- 観測演算は引数を取ることにより無限個の変数を表現することが可能であったが，制限により有限個の変数のみを使用する振舞仕様となる．有限個の変数に対する遷移規則は，C-TRASの持つ制約として無視した変数に依存しない．このため各変数の遷移規則は，元の振舞仕様で到達可能な状態から到達不能な状態へと遷移させることはない．また後述するが，有限個の変数に対する初期値は両仕様で一致する．
- 各演算のアリティにNatを取るとき，NzNatやZeroというサブソート上の変数を用いることができる．このような変数が現れる等式は，変数のソートをNatに置き換え，条件式 c を次のように変更する．

  変数Pのソートが NzNat である場合 : $P >= 1$ *and* c

  変数Pのソートが Zero である場合 : $P == 0$ *and* c

以上の理由により，操作演算・観測演算のアリティに可視ソートを取らない仕様に対して証明を行うことで，操作演算・観測演算のアリティに可視ソートを取る仕様でも同様に証明される．

操作演算・観測演算のアリティに可視ソートを取らない振舞仕様 $\mathcal{S}_B$ は次のような構成であり，一般に図4.13に示すように記述される．

- 隠蔽ソート H
- 可視ソート `Bool`, `Nat`, `NzNat`, `Zero`
- n個の観測演算 obs_i ($i = 1 .. n$)

  bop obs_i **:** H **->** V_i **.**

  可視ソート $V_i \in$ {`Bool`, `Nat`, `NzNat`, `Zero`} をコアリティにとる．
- m個の操作演算 act_j ($j = 1 .. m$)

bop act_j **:** H **->** H **.**

- 遷移規則を与える等式.

  ceq $obs_i(act_j(h))$ **=** v_{Tij} **if** c_{ij} **.**
  ceq $obs_i(act_j(h))$ **=** v_{Fij} **if not** c_{ij} **.**

  $^{\forall}i(1 \leq i \leq n)$ $^{\forall}j(1 \leq j \leq m)$ に対して，$^{\exists}h \in H$ で c_{ij} が満たされる時に，$obs_i(act_j(h))$ の値は v_{Tij} と等しい．同様に，c_{ij} が満たされない時の $obs_i(act_j(h))$ の値は v_{Fij} と等しい．ここで，v_{Tij}, v_{Fij}, c_{ij} は，操作演算を含まない任意の項である．条件無し等式は c_{ij} が true の場合と等しい.

  また各等式は，条件 c_{ij} が成り立つ場合，成り立たない場合に分けて記述しているが，条件分けは以下の条件を満たす任意のものが可能である.

  1. 全ての場合について観測演算の値が定義されること.
  2. 1 つの状態に対して異なる観測結果を二重に定義しないこと.

- 初期状態を与える定数.

  op $init$ **:** **->** H **.**

- 初期状態を定義する等式.

  ceq $obs_i(init)$ **=** v_{i0} **if** c_{i0} **.**

  $^{\forall}i(1 \leq i \leq n)$ に対して，c_{i0} が満たされる時，$obs_i(init)$ の値は v_{i0} と等しい．全ての等式で観測結果が定義されない時は，任意の値を許すことになる．ここで v_{i0}, c_{i0} は操作演算，観測演算を含まない任意の項である．また c_{i0} は命題であるため，初期状態に対する等式は各観測演算に対し 1 つあれば十分である．以下，初期値に対する等式は各観測演算に付き一つだけあるものとする.

- 安全性を定義する等式.

  ceq **safety** P **=** v_p **if** c_p **.**

  安全性を定義する P, v_p, c_p は操作演算を含まない任意の等式である.

振舞仕様 $\mathcal{S}_B$ と制限情報から，C-TRAS を用いて SMV 仕様に変換する．変換した SMV 仕様は図4.14のようになる．振舞仕様で P, c, v で表された項は，4.2.3項の項の変換で説明した方法により，それらと等価な SMV 式 P', c', v' へと変換される．ソート V から SMV 上の型 V' への変換は，制限情報と表4.3の変換規則により変換される．この変換処理を実装した関数 *toSMV* : 振舞仕様での項 -> SMV の式 . を用いて表すと，

```
mod! SAFETY{
  pr(BOOL)
  op safety _ : Bool -> Bool .
}

mod* S_B {
  pr(NAT + BOOL + SAFETY)
  *[ H ]*
  bop obs_1 : H -> V_1 .
  bop obs_2 : H -> V_2 .
        ⋮
  bop obs_n : H -> V_n .

  bop act_1 : H -> H .
  bop act_2 : H -> H .
        ⋮
  bop act_m : H -> H .
  op init : -> H .

  -- 遷移規則
  var h : H
  ceq obs_1(act_1(h)) = v_T11 if c_11 .
  ceq obs_1(act_1(h)) = v_F11 if not c_11 .
  ceq obs_1(act_2(h)) = v_T12 if c_12 .
  ceq obs_1(act_2(h)) = v_F12 if not c_12 .
        ⋮
  ceq obs_1(act_m(h)) = v_T1m if c_1m .
  ceq obs_1(act_m(h)) = v_F1m if not c_1m .

        ⋮

  ceq obs_n(act_1(h)) = v_Tn1 if c_n1 .
  ceq obs_n(act_1(h)) = v_Fn1 if not c_n1 .
  ceq obs_n(act_2(h)) = v_Tn2 if c_n2 .
  ceq obs_n(act_2(h)) = v_Fn2 if not c_n2 .
        ⋮
  ceq obs_n(act_m(h)) = v_Tnm if c_nm .
  ceq obs_n(act_m(h)) = v_Fnm if not c_nm .

  -- 初期値
  ceq obs_1(init) = v_10 if c_10 .
  ceq obs_2(init) = v_20 if c_20 .
        ⋮
  ceq obs_n(init) = v_n0 if c_n0 .

  -- 安全性
  ceq safety P = v_p if c_p .
}
```

図 4.13: 入力となる振舞仕様

```
MODULE main
VAR
  obs_1 : V'_1;
  obs_2 : V'_2;
    ⋮
  obs_n : V'_n;
  _action : {act_1, act_2, ... , act_m};

ASSIGN
  init(_action) := {act_1, act_2, ... , act_m};
  next(_action) := {act_1, act_2, ... , act_m};

  -- 遷移規則
  next(obs_1) := case
                   _action = act_1 & c'_11     : v'_T11;
                   _action = act_1 & !(c'_11)  : v'_F11;
                   _action = act_2 & c'_12     : v'_T12;
                   _action = act_2 & !(c'_12)  : v'_F12;
                     ⋮
                   _action = act_m & c'_1m     : v'_T1m;
                   _action = act_m & !(c'_1m)  : v'_F1m;
                esac;
   ⋮
  next(obs_n) := case
                   _action = act_1 & c'_n1     : v'_Tn1;
                   _action = act_1 & !(c'_n1)  : v'_Fn1;
                   _action = act_2 & c'_n2     : v'_Tn2;
                   _action = act_2 & !(c'_n2)  : v'_Fn2;
                     ⋮
                   _action = act_m & c'_nm     : v'_Tnm;
                   _action = act_m & !(c'_nm)  : v'_Fnm;
                esac;

  -- 初期状態
  init(obs_1) := case
                   c'_10 : v'_10;
                   1     : { V'_1 に含まれる全ての値 };
                 esac;
    ⋮
  init(obs_n) := case
                   c'_n0 : v'_n0;
                   1     : { V'_n に含まれる全ての値 };
                 esac;

-- 安全性
SPEC AG c' -> (P'_p = v'_p)
```

図 4.14: 入力となる振舞仕様と制限情報より変換した SMV 仕様

$$toSMV(c_{ij}) \equiv c'_{ij} \quad toSMV(c_p) \equiv c'_p \quad toSMV(c_{i0}) \equiv c'_{i0}$$
$$toSMV(v_{Tij}) \equiv v'_{Tij} \quad toSMV(v_p) \equiv v'_p \quad toSMV(v_{i0}) \equiv v'_{i0}$$
$$toSMV(v_{Fij}) \equiv v'_{Fij} \quad toSMV(P) \equiv P' \quad toSMV(V_i) \equiv V'_i$$

ただし演算 $\equiv$ の定義は次のように与えられる．SMV 式上の演算 $=$ を用い，定数項 T_c に対して，$toSMV(T_c) = T'_c \Leftrightarrow toSMV(T_c) \equiv T'_c$ である．また，n 個の変数・観測演算を含む項 T と SMV 式 T' が $toSMV(T) \equiv T' \Leftrightarrow^{\forall} i(1 \le i \le n)$ に対し，その引数となる項 arg_i とそれに対応する SMV 式 arg'_i が $toSMV(arg_i) \equiv arg'_i$ である時に，

$$toSMV(T(arg_1, arg_2, \ldots, arg_n)) \equiv T'(arg'_1, arg'_2, \ldots, arg'_n)$$

であることをいう．

である．また振舞仕様での観測演算名 obs_i・操作演算名 act_j は，SMV 仕様でも変数名，列挙型の名前として使用する．

図4.14の振舞仕様に対しモデル検査を行った結果，前提より安全性に対し表4.5に示す長さ k の反例が報告される．a_i は反例の遷移系列の i 番目の状態での__action の値であり，$^{\forall}i(1 \le i \le k)$ に対し，$a_i \in \{act_j \mid 1 \le j \le m\}$ である．また同様に $o_{j,i}$ は反例の遷移系列の i 番目の状態での obs_j の観測結果であり，$^{\forall}i(1 \le i \le k)\ ^{\forall}j(1 \le j \le n)$ に対し，$o_{j,i} \in \{V'_j$に含まれる全ての値$\}$である．

表 4.5: C-TRAS により変換した振舞仕様に対するモデル検査結果の反例

	1	2	…	i	$i+1$	…	k
__action	a_1	a_2	…	a_i	a_{i+1}	…	a_k
obs_1	$o_{1,1}$	$o_{1,2}$	…	$o_{1,i}$	$o_{1,i+1}$	…	$o_{1,k}$
obs_2	$o_{2,1}$	$o_{2,2}$	…	$o_{2,i}$	$o_{2,i+1}$	…	$o_{2,k}$
⋮	⋮	⋮		⋮	⋮	⋱	⋮
obs_n	$o_{n,1}$	$o_{n,2}$	…	$o_{n,i}$	$o_{n,i+1}$	…	$o_{n,k}$

SMV で発見された反例が，振舞仕様 $\mathcal{S}_B$ でも反例となっていることを示すために，以下の 2 点を証明する．

1. 反例で示された遷移系列が，元の振舞仕様でも同順で操作演算を適用することにより，同様の遷移系列が得られること．

2. 反例で示された最後の状態に対し，CafeOBJ による検証で安全性を満たさないことが示されること．

証明.

まず遷移系列の初期状態が，振舞仕様 $\mathcal{S}_B$ の初期状態のひとつであることを示す．SMV の初期値に関する条件式は 2 つに場合わけされる．

- c'_{i0} が FALSE である．c'_{i0} と等価な振舞仕様の項 c_{i0} も false であり，$obs_i(init)$ は V_i 上の任意の項を取ることができる．このため，ただちに obs_i の初期値 $o_{i,1}$ は振舞仕様 $\mathcal{S}_B$ でも観測しうる値である．
- c'_{i0} が TRUE であり，$o_{i,1} = v'_{i0}$ である．c'_{i0} が TRUE であることから，振舞仕様 $\mathcal{S}_B$ でも c_{i0} の値は true となる．よって，振舞仕様 $\mathcal{S}_B$ では，初期状態に対する obs_i による観測結果は v_{i0} となる．$toSMV(v_{i0}) \equiv v'_{i0}$ より，$o_{i,1}$ は振舞仕様 $\mathcal{S}_B$ でもとりうる値である．

それぞれの場合について，変数 obs_i の $o_{i,1}$ の値が，振舞仕様の初期状態に対する obs_i の観測結果と一致する．これは $^{\forall}i(1 \le i \le n)$ に対し成り立つため，SMV が挙げたの反例の初期状態と，振舞仕様 $\mathcal{S}_B$ の初期状態は等しい．

次に，SMV が挙げた反例の i 番目の状態に振舞仕様でも到達可能であると仮定し，i+1 番目の状態に振舞仕様で到達可能であることを示す．振舞仕様 $\mathcal{S}_B$ に対して前提より，$^{\exists}h_i \in H$ が存在し，$^{\forall}j(1 \le j \le n)\ obs_j(h_i) = o_{j,i}$ である．

まず，操作演算は常に任意のものを選択することができるため，i+1 番目において，a_{i+1} は選択することができる．

i 番目の状態から遷移した i+1 番目の状態でとる obs_j の値を求める．反例の i 番目の状態において，反例での__action の値は $^{\exists}x(1 \le x \le m)\ a_i = act_x$ である．obs_j に関する遷移規則のうち，__action = act_x の時に適用できる規則は 2 つある．

```
__action = act_x & c'_jx     : v'_Tjx;
__action = act_x & !(c'_jx)  : v'_Fjx;
```

i 番目の状態において c'_{jx} を満たすならば v'_{Tjx} が選択され，満たさないならば v'_{Fjx} が SMV によって選択される．この状況において振舞仕様 $\mathcal{S}_B$ で適用される等式は，左辺が $obs_j(act_x(h))$ である等式であり，以下の 2 つが宣言されている．

```
ceq obs_j(act_x(h)) = v_Tjx if c_jx .
ceq obs_j(act_x(h)) = v_Fjx if not c_jx .
```

明らかに，$toSMV(c_{jx}) \equiv c'_{jx}$，$toSMV(v_{Tjx}) \equiv v'_{Tjx}$，$toSMV(v_{Fjx}) \equiv v'_{Fjx}$ である．また双方の i 番目の状態に対する観測結果は前提より等しいため，SMV の反例の i 番目の

状態における c'_{jx} の値と，振舞仕様 $\mathcal{S}_B$ の h_i の状態に対する c_{jx} の値は一致する．同様に，v_{Tjx} と v'_{Tjx}，v_{Fjx} と v'_{Fjx} の値も一致することが言える．このため，i+1 番目の状態において obs_j の値は両仕様で等しい．同様の理由により，$^\forall j(1 \leq j \leq r)$ に対し，$toSMV(obs_j(act_x(h_i)))$ と SMV 上の変数 obs_j の値は等しい．

ここでは 2 つの遷移規則があると仮定し議論を進めたが，より適切には次のように r 個の遷移規則があるとする．

$$
\begin{array}{l}
\quad\vdots \\
\texttt{ceq}\ obs_j(act_x(h))\ \texttt{=}\ v^1_{jx}\ \texttt{if}\ c^1_{jx}\ \texttt{.} \\
\texttt{ceq}\ obs_j(act_x(h))\ \texttt{=}\ v^2_{jx}\ \texttt{if}\ c^2_{jx}\ \texttt{.} \\
\quad\vdots \\
\texttt{ceq}\ obs_j(act_x(h))\ \texttt{=}\ v^r_{jx}\ \texttt{if}\ c^r_{jx}\ \texttt{.} \\
\quad\vdots
\end{array}
$$

これらの等式に対し，それと等価な SMV の遷移規則が次のように生成される．

$$
\begin{array}{l}
\quad\vdots \\
\texttt{\_action}\ \texttt{=}\ act_x\ \texttt{\&}\ c'^1_{jx}\ \texttt{:}\ v'^1_{jx}\texttt{;} \\
\texttt{\_action}\ \texttt{=}\ act_x\ \texttt{\&}\ c'^2_{jx}\ \texttt{:}\ v'^2_{jx}\texttt{;} \\
\quad\vdots \\
\texttt{\_action}\ \texttt{=}\ act_x\ \texttt{\&}\ c'^r_{jx}\ \texttt{:}\ v'^r_{jx}\texttt{;} \\
\quad\vdots
\end{array}
$$

また振舞仕様の項から SMV 式の生成方法が，意味的に等しい式を生成するため $^\forall \ell\,(1 \leq \ell \leq r)$ に対し，$toSMV(c^\ell_{ij}) \equiv c'^\ell_{ij}$, $toSMV(v^\ell_{ij}) \equiv v'^\ell_{ij}$ である．このため，SMV での変数の値と，振舞仕様での観測結果が一致する状態では，それぞれの式のとる値は等しい．よって，$toSMV(obs_j(act_x(h_i)))$ と SMV 上の変数 obs_j の値は等しい．よって $^\forall j(1 \leq j \leq r)$ に対しても同様のことが言えるため，任意の場合わけの遷移規則についても i 番目の状態が等しいときに，等しい操作演算を行うと等しい状態に遷移する．

以上により，SMV によって得られた反例が示す状態遷移系列は，振舞仕様でも同様に遷移することが示された．よって初期状態から反例が得られた状態まで振舞仕様においても到達することが示された．

次に反例の最終状態に対して，CafeOBJ により検証を行った結果，安全性を満たさないことが証明されることを示す．

SMV が安全性 AG(c'_p -> ($P' = v'_p$)) に対して反例を挙げたことから，反例の k 番目の状態は c'_p -> ($P' = v'_p$) を満たさない．$toSMV(P) \equiv P'$, $toSMV(c_p) \equiv c'_p$, $toSMV(v_p) \equiv v'_p$ であり，反例の k 番目の状態と等しい状態 $h_k \in H$ が存在する．よって，CafeOBJ により安全性 c_p -> ($P = v_p$) の項を状態 h_k で簡約すると false が得られる．

証明 了.

CafeOBJ では，ある遷移規則が適用可能かどうかや遷移後どのような状態に遷移するかを簡約により計算する．CafeOBJ は項の簡約が適用可能な場所を探すために，まず部分項と等式の左辺がパターンマッチングを行い，一致する個所を探す．次に等式に条件がある場合，部分項が等式の条件を満たすかどうか，if 節の項を簡約をすることで検査する．条件を満たす場合，もしくは条件が無い場合，等式の左辺へと簡約が行われる．簡約順序によらず得られる結果が常に同一となる性質を合流性といい，簡約が必ず停止する性質を停止性という．この 2 つの性質を満たす仕様を完備な仕様という．BOOL, NAT モジュール上で定義されている演算による項は，完備である事が分かっている．このため，簡約可能な等式が複数ある場合でも，適用順序は結果に影響しない．

SMV 仕様では，最小の動作ステップは 1 回の状態遷移である．図4.10で示しように，遷移を行うために，振舞仕様での操作演算名の等価性と，サブソート関係を満たすかどうかの検査を行う．これは，部分項と等式の左辺の一致を検査するパターンマッチの処理に相当する．次に条件付き等式の場合，条件式を満たすかどうか計算する．SMV による条件式の計算は，if 節の項を繰り返し簡約することに対応する．またその時の計算順序は，最左最内戦略に基づく簡約と等しい[1]．図4.9の例では，SMV は <，state(S,P)，=，and の順に計算を行う．実際の CafeOBJ の簡約順と SMV の計算順は異なる可能性があるが，合流性により条件式の評価は常に等しい結果が得られることが保証される．

このため抽象データ型が完備である仕様に対して，範囲外に遷移する場合を除き，制限した振舞仕様上で可能な状態の遷移系列と SMV 仕様上で可能な状態の遷移系列は一致する．

4.4 CafeOBJ 仕様変換器 C-TRAS を用いた検査法

図4.15に両検査器を用いた検証のワークフローを示す．まずはじめに振舞仕様を記述する．記述した振舞仕様に対して制限情報を与え，C-TRAS により仕様を変換することでモデル検査を行う．

- モデル検査による検証

  モデル検査結果に対する解釈は次のようになる．

  - 反例が発見された場合，それは元の振舞仕様にも誤りが存在する．反例より誤りに至る状態遷移系列が分かる．

[1] FALSE & X や TRUE | X のように途中で式全体の値が決まり，最内ではない場合もある．このような場合であっても合流性により等価である．

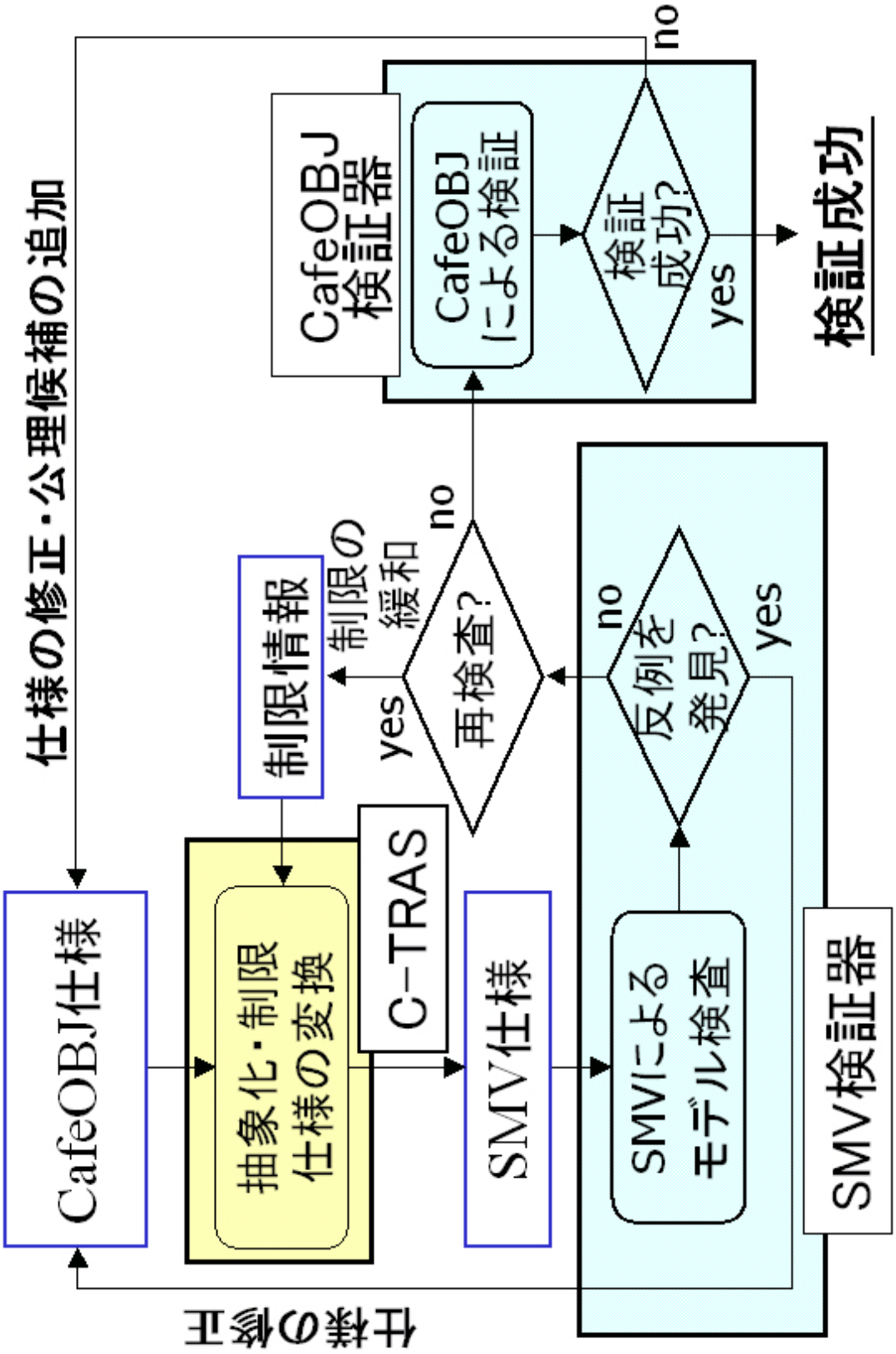

図 4.15: C-TRAS を用いた検査法のフロー図

1) 仕様中の軽微な誤りの場合，反例より容易に仕様が修正できる．修正後，再びも出る検査を行う．
2) 仕様中の重大な誤りの場合，反例よりその原因を特定し仕様の修正を試みる．この作業は困難であり，誤りを修正した仕様を作成できないかもしれない．
3) 実現不能な問題をとこうとしている場合．仕様の修正を繰り返しても，反例を取り除くことはできない．

1, 2 の場合，反例より仕様の修正が容易になる．反例が自動で得られることから，証明論敵手法に比べ早期に誤りを発見し，修正することができる．

- 反例が発見されない場合，その原因は二つ考えられる．

  1) 制限によって切り捨てた状態遷移機械に反例が含まれている．
  2) 振舞仕様に誤りが存在しないため，モデル検査で反例が得られない．

  制限によって得られた SMV 仕様では誤りが無いことを示すことはできない．このためより強い確信を得たい場合は，制限を緩め再びモデル検査を行う．十分な確信が得られた時点で，CafeOBJ 検証器による検証に切り替える．

- CafeOBJ 検査器による検証

  無限状態に対する証明はモデル検査器では直接行う事ができない．このため CafeOBJ を用いて検証を行う．証明論的手法による検証は，公理と仕様から証明できることは限られており，多くの場合 検証の途中で定理を追加する必要がある．

  定理を追加するためには，その定理が成り立つことを振舞仕様上で成り立つことを証明しなければならない．この作業は容易ではなく，検証者の能力に大きく依存する．しかしモデル検査によって定理の証明を支援することができる．定理の候補をモデル検査で検証することにより，誤りのある定理候補の証明を試みる事が無くなる．定理を証明するために別の定理が必要になることもあり，モデル検査によって候補を絞って検証できることは能率的な検証に大きく貢献する．

振舞仕様に対しモデル検査技術を適用することは，制限情報を与えること以外，自動で行う事ができる．また誤りがある場合に反例を示すことができる点で CafeOBJ による検証を支援することができる．

両者を併用して検証を行うことにより，誤りの発見と仕様の修正・検証を機械と人で分担し，検証を能率良く行う事が期待される．

第 5 章

例題

本章ではいくつかの例題に対し C-TRAS を利用してモデル検査を行う事で，事前検査の有効性を確かめる．

5.1 相互排他問題

相互排他問題として Load と Store に基づく方法，Test & Set に基づく方法，Peterson のアルゴリズムを取り上げる．

5.1.1 Load と Store に基づく相互排他システム

Load と Store に基づく相互排他システムは，ある変数の値を読み，値がある条件検査し (test)，条件を満たす時は次に値の変更をする (set)．test と set のそれぞれの命令は，2 つの演算に分けることができない不可分な命令であるとする．このような命令をを原始的 (atomic) な操作という．

```
0: < Local な処理 >
       ↓ rem
1:  < 待機状態 >
       ↓ test
      資源はロックされていない 2 へ．資源はロックされている 1 へ．
2:  < 資源の獲得動作 >
       ↓ set
      共有資源をロックし，3 へ．
3:  < Critical Section>
       ↓ leave
     Local な処理へ戻る 0 へ．
```

図 5.1: Load と Store に基づく相互排他システムの各プロセスの動作

図5.1に示したプロセスは，0〜3 の 4 状態を持ち，共有資源がロックされている状態とされていない状態で 2 つの状態持つ．このためプロセス数を n とすると単純に計算して，可能な状態数は 2×4^n 個である．当然 n が非常に大きくなれば，実質無限となる．各プロセスは，状態 0 では自分の持つ計算資源でできる処理を行う．他の共有資源が必要になった時点で rem を行い，共有資源獲得待ち状態に遷移する．test を行い，誰も資源を獲得していない状態であったら，次に set を行い資源をロックし，クリティカルセクションへ遷移する．クリティカルセクションを抜ける時は leave を行い，共有資源を解放して状態 0 に戻る．この動作を全てのプロセスが繰り返し行う．プロセスの実行権は，プロセスが 0, 1, 2, 3 の各状態にいる時インタリーブによって他プロセスに実行権を移すことができる．この時のスケジューリングは任意のものを考える．

図5.2の振舞仕様に対し，2 プロセスに制限して C-TRAS により SMV 仕様に変換した (図5.3)．状態数は $2\times4^2=32$ 個[1]と非常に少ない場合について検証する．

図5.3の SMV 仕様に対し，Cadence SMV でモデル検査を行った．その結果，表5.1のような反例が得られた．

表 5.1: Load と Store に基づく相互排他システムに対するモデル検査結果

	1	2	3	4	5	6	7
__action	rem	test	rem	test	set	set	leave
_nat0	1	1	0	0	1	0	0
lock	FALSE	FALSE	FALSE	FALSE	FALSE	TRUE	TRUE
st[0]	0	0	0	1	2	2	3
st[1]	0	1	2	2	2	3	3

表5.1は縦軸に変数，横軸に時間軸を取った票である．1 の列が初期状態で，7 の列が反例が得られた状態である．振舞仕様では lock, st[0], st[1] の 3 変数の組で状態が表されるが，SMV では次に実行する操作演算とその引数 __action, _nat0 が状態変数に追加される．各列の __action, _nat0 は次に実行される操作演算とその引数なので注意して読んでいただきたい．

この反例は，`AG((st[0] = 3 & st[1] = 3) ->(0 = 1)) = TRUE` に対する反例である．この反例は，最も少ない遷移回数で現れる反例の一つとなっており，反例の理解を容易にする．Cadence SMV では，反例が得られた時点で処理を打ち切るので反例を示さないが，実際には `AG ((st[1] = 3 & st[0] = 3) ->(1 = 0)) = TRUE` の安全性につい

[1]SMV 仕様に変換すると $2\times4^2\times4\times2=256$ 状態になる．

```
mod! SAFETY{
  pr(BOOL)
  op safety _ : Bool -> Bool {prec: 100}
}

mod* TestOrSet {
  pr(NAT + SAFETY)
  *[Mutex]*

-- observations
  bop lock : Mutex     -> Bool .
  bop st   : Mutex Nat -> Nat . ** limit * 2 -> 4

-- actions
  bop rem   : Mutex Nat -> Mutex . ** limit * 2 -> *
  bop test  : Mutex Nat -> Mutex . ** limit * 2 -> *
  bop set   : Mutex Nat -> Mutex . ** limit * 2 -> *
  bop leave : Mutex Nat -> Mutex . ** limit * 2 -> *

-- initial state
  op ini : -> Mutex

-- CafeOBJ variables
  var  S   : Mutex
  vars I J : Nat

-- for rem
  eq  lock(rem(S,I)) = lock(S) .
  ceq st(rem(S,I),J) = 1         if     I == J and st(S,I) == 0 .
  ceq st(rem(S,I),J) = st(S,J)   if not(I == J and st(S,I) == 0) .

-- for test
  eq  lock(test(S,I)) = lock(S) .
  ceq st(test(S,I),J) = 2        if     I == J and st(S,I) == 1 and not lock(S) .
  ceq st(test(S,I),J) = st(S,J)  if not(I == J and st(S,I) == 1 and not lock(S)) .

-- for set
  ceq lock(set(S,I))  = true     if     st(S,I) == 2 .
  ceq lock(set(S,I))  = lock(S)  if not(st(S,I) == 2) .
  ceq st(set(S,I),J)  = 3        if     I == J and st(S,I) == 2 .
  ceq st(set(S,I),J)  = st(S,J)  if not(I == J and st(S,I) == 2) .

-- for leave
  ceq lock(leave(S,I)) = false   if     st(S,I) == 3 .
  ceq lock(leave(S,I)) = lock(S) if not(st(S,I) == 3) .
  ceq st(leave(S,I),J) = 0       if     I == J and st(S,I) == 3 .
  ceq st(leave(S,I),J) = st(S,J) if not(I == J and st(S,I) == 3) .

-- initial state
  eq st(ini,I) = 0 .
  eq lock(ini) = false .

-- safety property
  eq safety (st(S,I) == 3 and st(S,J) == 3) implies I == J = true .
}
```

図 5.2: Load と Store に基づく相互排他システムに対する振舞仕様

```
MODULE main

VAR
  st : array 0 .. 1 of 0 .. 3;
  lock : boolean;
  _nat0 : 0 .. 1;
  __action : {rem, test, set, leave};

ASSIGN
  init(st[0]) := 0;
  init(st[2]) := 0;
  init(lock)  := FALSE;

  next(lock) :=
    case
      __action = leave  & !st[_nat0] = 3 : lock;
      __action = leave  & st[_nat0] = 3 : FALSE;
      __action = set  & !st[_nat0] = 2 : lock;
      __action = set  & st[_nat0] = 2 : TRUE;
      __action = test  : lock;
      __action = rem  : lock;
    esac;

  next(st[0]) :=
    case
      __action = leave  & !(_nat0 = 0 & st[_nat0] = 3) : st[0];
      __action = leave  & _nat0 = 0 & st[_nat0] = 3 : 0;
      __action = set  & !(_nat0 = 0 & st[_nat0] = 2) : st[0];
      __action = set  & _nat0 = 0 & st[_nat0] = 2 : 3;
      __action = test  & !(_nat0 = 0 & st[_nat0] = 1 & !lock) : st[0];
      __action = test  & _nat0 = 0 & st[_nat0] = 1 & !lock : 2;
      __action = rem  & !(_nat0 = 0 & st[_nat0] = 0) : st[0];
      __action = rem  & _nat0 = 0 & st[_nat0] = 0 : 1;
    esac;

  next(st[1]) :=
    case
      __action = leave  & !(_nat0 = 1 & st[_nat0] = 3) : st[1];
      __action = leave  & _nat0 = 1 & st[_nat0] = 3 : 0;
      __action = set  & !(_nat0 = 1 & st[_nat0] = 2) : st[1];
      __action = set  & _nat0 = 1 & st[_nat0] = 2 : 3;
      __action = test  & !(_nat0 = 1 & st[_nat0] = 1 & !lock) : st[1];
      __action = test  & _nat0 = 1 & st[_nat0] = 1 & !lock : 2;
      __action = rem  & !(_nat0 = 1 & st[_nat0] = 0) : st[1];
      __action = rem  & _nat0 = 1 & st[_nat0] = 0 : 1;
    esac;

  init(_nat0) := {0, 1};
  next(_nat0) := {0, 1};
  init(__action) := {rem, test, set, leave};
  next(__action) := {rem, test, set, leave};

SPEC AG ((st[0] = 3 & st[0] = 3) -> (0 = 0)) = TRUE
SPEC AG ((st[0] = 3 & st[1] = 3) -> (0 = 1)) = TRUE
SPEC AG ((st[1] = 3 & st[0] = 3) -> (1 = 0)) = TRUE
SPEC AG ((st[1] = 3 & st[1] = 3) -> (1 = 1)) = TRUE
```

図 5.3: Load と Store に基づく相互排他システムの SMV 仕様

ても反例が得られる[2].

では表5.1に示した反例が，本当に反例となっているか CafeOBJ で確かめる．図5.4に反例を確かめる proof スコアを示す.

```
open TestOrSet
red lock(ini) == false .
red st(ini,0) == 0 .
red st(ini,1) == 0 .
close .

open TestOrSet
red lock(rem(ini,1)) == false .
red st(rem(ini,1),0) == 0 .
red st(rem(ini,1),1) == 1 .
close .

open TestOrSet
op a2 : -> Mutex .
eq lock(a2) = false .
eq st(a2,0) = 0 .
eq st(a2,1) = 1 .
red lock(test(a2,1)) == false .
red st(test(a2,1),0) == 0 .
red st(test(a2,1),1) == 2 .
close .

open TestOrSet
op a3 : -> Mutex .
eq lock(a3) = false .
eq st(a3,0) = 0 .
eq st(a3,1) = 2 .
red lock(rem(a3,0)) == false .
red st(rem(a3,0),0) == 1 .
red st(rem(a3,0),1) == 2 .
close .

open TestOrSet
op a4 : -> Mutex .
eq lock(a4) = false .
eq st(a4,0) = 1 .
eq st(a4,1) = 2 .
red lock(test(a4,0)) == false .
red st(test(a4,0),0) == 2 .
red st(test(a4,0),1) == 2 .
close .
```

```
open TestOrSet
op a5 : -> Mutex .
eq lock(a5) = false .
eq st(a5,0) = 2 .
eq st(a5,1) = 2 .
red lock(set(a5,1)) == false .
red st(set(a5,1),0) == 2 .
red st(set(a5,1),1) == 3 .
close .

open TestOrSet
op a6 : -> Mutex .
eq lock(a6) = true .
eq st(a6,0) = 2 .
eq st(a6,1) = 3 .
red lock(set(a6,0)) == true .
red st(set(a6,0),0) == 3 .
red st(set(a6,0),1) == 3 .
close .

open TestOrSet
op a7 : -> Mutex .
eq lock(a7) = true .
eq st(a7,0) = 3 .
eq st(a7,1) = 3 .
red ((st(a7,0) == 3 and st(a7,1) == 3)
     implies 0 == 1) == true .
close .
```

図 5.4: Load と Store に基づく相互排他システムの反例に対する CafeOBJ での検証

proof スコアは，観測結果と反例の lock, st[0], st[1] に示す値が一致するか，初期状態からはじめ，それぞれの状態に表に示す操作演算を実行した結果，次の状態でも成り立つか検査する．状態 7 まで順に検査し全て true ならば 状態 7 まで確かに到達する経路があることがわかる．また状態 7 において安全性を検証し，安全性が偽となり満たさないことを

[2]配列中の変数を展開することで，NuSMV でも検証可能.

確認する．実際に CafeOBJ で実行した結果，最後の安全性以外全て true が返り，安全性は false が得られた．これにより，初期状態から 7 の状態までの遷移系列は確かに存在し，7 の状態は安全性に対する反例であることが分かった．

Load と Store に基づく図5.1のプロセス動作では，相互排他を保障できないことがわかった．反例に対し十分検討を加えると，プロセス１が test に成功し，次に set を行う前にプロセスの実行権が移り，プロセス 0 が状態 2 まで遷移していることがわかる．状態２では，値をセットした無条件にクリティカルセクションに入るため，５の状態に到達することが問題であることがわかる．test を行ってから， set を行うまでを一つの原始的な演算とすると，この反例が現れないように思われる．

5.1.2 Test & Set に基づく相互排他

Load と Store に基づく相互排他システムの反例に対する知見から，test と set を一つの原始的な演算 testandset として行うように変更する．Load と Store 命令は CPU で原始的な演算として扱われている．Test & Set は，Sparc プロセッサにおいて CPU 上に実際に実装されている命令であり，Sparc プロセッサにおいても原始的な演算として機能する．

```
0: < Local な処理 >
       ↓ rem
1:  < 待機状態 >
       ↓ test&set
      資源を獲得できた場合 2 へ．獲得できない場合 1 で待機
2:  < Critical Section>
       ↓ leave
     Local な処理へ戻る
```

図 5.5: Test & Set に基づく相互排他システムの各プロセスの動作

図5.5の動作をするプロセスで，Load と Store に基づく相互排他システムと同じように振舞仕様を記述した (図5.6)．操作演算 test と set の替わりに testandset を追加した．これにより各プロセスの状態数は 0, 1, 2 の３状態となる．

図5.5を元に記述した振舞仕様を図5.6に示す．操作演算 rem, testandset, leave によりプロセスが遷移し，観測演算 state, lock で状態を観測する．プロセスの状態はアルゴリズム中に記述した 0, 1, 2 と state で観測されるプロセスの状態 0, 1, 2 は直接対応する．安全性は，クリティカルセクションである状態 2 に異なる 2 つのプロセスはいないことを記述した．

この仕様を C-TRAS に与え，SMV 仕様に変換したものを図5.7に示す．

有限化により，プロセス数は 8 個に制限した．このため，状態空間は $2 \times 3^8 = 13122$ 状

```
mod! SAFETY{
  pr(BOOL)
  op safety _ : Bool -> Bool {prec: 100}
}
mod* TestAndSet {
  pr(NAT + SAFETY)
  *[Sys]*

-- observations
  bop lock : Sys     -> Bool .
  bop st   : Sys Nat -> Nat  .        ** limit * 8 -> 3

-- actions
  bop rem   : Sys Nat -> Sys .        ** limit * 8 -> *
  bop testandset : Sys Nat -> Sys .   ** limit * 8 -> *
  bop leave : Sys Nat -> Sys .        ** limit * 8 -> *

-- initial state
  op ini : -> Sys

-- CafeOBJ variables
  var S : Sys
  vars I J : Nat

-- for rem
  eq  lock(rem(S,I)) = lock(S) .
  ceq st(rem(S,I),J) = 1         if     I == J and st(S,I) == 0 .
  ceq st(rem(S,I),J) = st(S,J)   if not(I == J and st(S,I) == 0) .

-- for testandset
  ceq lock(testandset(S,I)) = true     if     st(S,I) == 1 and not lock(S) .
  ceq lock(testandset(S,I)) = lock(S)  if not(st(S,I) == 1 and not lock(S)) .
  ceq st(testandset(S,I),J) = 2        if     I == J and st(S,I) == 1 and not lock(S) .
  ceq st(testandset(S,I),J) = st(S,J)  if not(I == J and st(S,I) == 1 and not lock(S)) .

-- for leave
  ceq lock(leave(S,I)) = false   if     st(S,I) == 2 .
  ceq lock(leave(S,I)) = lock(S) if not(st(S,I) == 2) .
  ceq st(leave(S,I),J) = 0       if     I == J and st(S,I) == 2 .
  ceq st(leave(S,I),J) = st(S,J) if not(I == J and st(S,I) == 2) .

-- initial state
  eq st(ini,I) = 0 .
  eq lock(ini) = false .

-- safety property
  eq safety (st(S,I) == 2 and st(S,J) == 2) implies (I == J) = true .
}
```

図 5.6: Test & Set に基づく相互排他システムの振舞仕様

```
MODULE main

VAR
  st : array 0 .. 7 of 0 .. 2;
  lock : boolean;
  _nat0 : 0 .. 7;
  __action : rem, testandset, leave;

ASSIGN
  init(st[0]) := 0;
     ⋮
  init(st[7]) := 0;
  init(lock)  := FALSE;

  next(lock) :=
    case
      __action = leave  & !st[_nat0] = 2 : lock;
      __action = leave  & st[_nat0] = 2 : FALSE;
      __action = testandset  & !(st[_nat0] = 1 & !lock) : lock;
      __action = testandset  & st[_nat0] = 1 & !lock : TRUE;
      __action = rem  : lock;
    esac;

  next(st[0]) :=
    case
      __action = leave  & !(_nat0 = 0 & st[_nat0] = 2) : st[0];
      __action = leave  & _nat0 = 0 & st[_nat0] = 2 : 0;
      __action = testandset  & !(_nat0 = 0 & st[_nat0] = 1 & !lock) : st[0];
      __action = testandset  & _nat0 = 0 & st[_nat0] = 1 & !lock : 2;
      __action = rem  & !(_nat0 = 0 & st[_nat0] = 0) : st[0];
      __action = rem  & _nat0 = 0 & st[_nat0] = 0 : 1;
    esac;

    ⋮

  next(st[7]) :=
    case
      __action = leave  & !(_nat0 = 7 & st[_nat0] = 2) : st[7];
      __action = leave  & _nat0 = 7 & st[_nat0] = 2 : 0;
      __action = testandset  & !(_nat0 = 7 & st[_nat0] = 1 & !lock) : st[7];
      __action = testandset  & _nat0 = 7 & st[_nat0] = 1 & !lock : 2;
      __action = rem  & !(_nat0 = 7 & st[_nat0] = 0) : st[7];
      __action = rem  & _nat0 = 7 & st[_nat0] = 0 : 1;
    esac;

  init(_nat0) := 0, 1, 2, 3, 4, 5, 6, 7;
  next(_nat0) := 0, 1, 2, 3, 4, 5, 6, 7;

  init(__action) := rem, testandset, leave;
  next(__action) := rem, testandset, leave;

SPEC AG ((st[0] = 2 & st[0] = 2) -> (0 = 0)) = TRUE
SPEC AG ((st[0] = 2 & st[1] = 2) -> (0 = 1)) = TRUE
              ⋮
SPEC AG ((st[7] = 2 & st[7] = 2) -> (7 = 7)) = TRUE
```

図 5.7: Test & Set に基づく相互排他システムの SMV 仕様

態に制限される．また，安全性は $8 \times 8 = 64$ 個生成された[3]．この仕様に対して Cadence SMV でモデル検査を行った結果，64 個の安全性に対してモデル検査を行った．検査の結果，全ての安全性に対して true という結果が得られた．これにより 8 プロセスまでは相互排他を保存し，同時に共有資源を獲得しないことが分かった．ただし，これが任意のプロセス数で成り立つかどうかについては分からない．そのためには，振舞仕様に対し CafeOBJ で検証をする必要がある．しかし，Load と Store に基づく相互排他システムの場合とは異なり，モデル検査により反例が得られなかった．モデル検査を用いることで，Load と Store に基づく相互排他システムの安全性を示そうとする労力を省くことができた．また反例から仕様修正のヒントを得て，モデル検査では用意には反例を見つけることができない相互排他システムを設計することができた．

次に，Peterson のアルゴリズムを取り上げる．

5.1.3 Peterson のアルゴリズム

Peterson のアルゴリズムでの各プロセスの動作を簡単に表すと図5.8の 0 から 4 の動作になる．またフローチャートで表したものを図5.9に示す．プログラムリスト左の 0〜4 の数字と，フローチャートの各状態中の 0〜4 の数字はそれぞれ対応している．

```
0:  flag[i] := true
1:  turn := i
    while(
2:    flag[~i] and
3:      turn = i)
  <Critical Section>
4:  flag[i] := false

   ただし i : {0,1}
          flag[i] : {true, false}
          ~0 = 1, ~1 = 0
   とする.
```

図 5.8: Peterson のアルゴリズム

[3]64 個中 8 個は，状態遷移機械によらず恒真である．

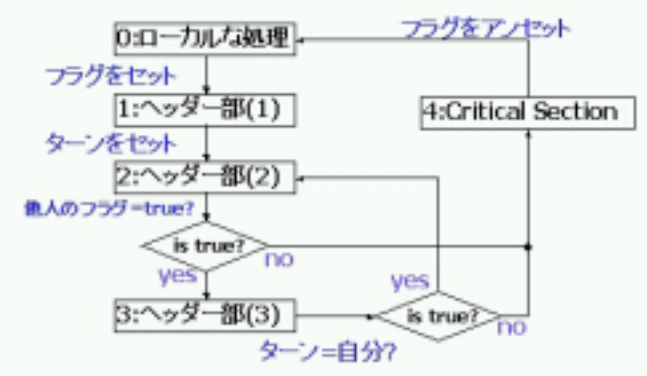

図 5.9: 各プロセスの動作

この仕様で登場するものには，２つのプロセスと１つの共有資源がある．また相互排他を実現するために共有変数である turn と，各プロセスに一つずつ flag がある．

初期状態では，２つのプロセスは 0 の命令の実行前の状態であり，turn は 0 に設定されているとする．プロセスが 0 の状態にいる時は共有資源を必用としない処理を行っている状態に相当する．プロセスが共有資源を使用したいときは，0) 自分のフラグをセットする．1) 共有変数 turn の値を自分のプロセス ID にセットする．という二つの操作を行う．ここまでを処理を行うと，共有資源の獲得待ちの状態になる．他プロセスが共有資源を獲得しようとしていない時は，2) の条件から直ちに Critical Section に入ることができる．2 つのプロセスが同時に Critical Section に入ろうとした時は，3) の処理により先に turn を設定したプロセスが Critical Section に入ることができる．残りのプロセスは while ループを実行し続け，他のプロセスが Critical Section を抜けるまで入ることはできない．Critical Section を抜けたプロセスは，4) フラグを下ろし，Critical Section にいないことを知らせる．これにより，待機していたプロセスが共有資源を獲得することができる．

この Peterson のアルゴリズムは，プロセスの数を２つに限定した相互排他問題に対する１つの解である．実行が開始されてから終了するまでの間に他の処理が割り込まない処理をアトミック (atomic) な処理と呼ぶ．Peterson のアルゴリズムでは，以下の５つの処理をアトミックな処理として扱っている．

- flag を true にセットする処理.
- turn に 0 または 1 をセットする処理.
- 他プロセスの flag の値をチェックする処理.
- turn の値を check する処理.

- flag を flase にセットする処理.

相互排他問題の検証をする上で，アトミックな処理をどのように選択するかは非常に重要である.

Peterson のアルゴリズムに対する振舞仕様

ここでは先に示したの５つの処理をアトミックな処理として扱い，図5.10のようにモデル化した．これを元に，図5.11のように振舞仕様を記述した．またここでは一度 Critical Section を抜けたプロセスは初期状態に戻り，共有資源の獲得を再度試みるようにした.

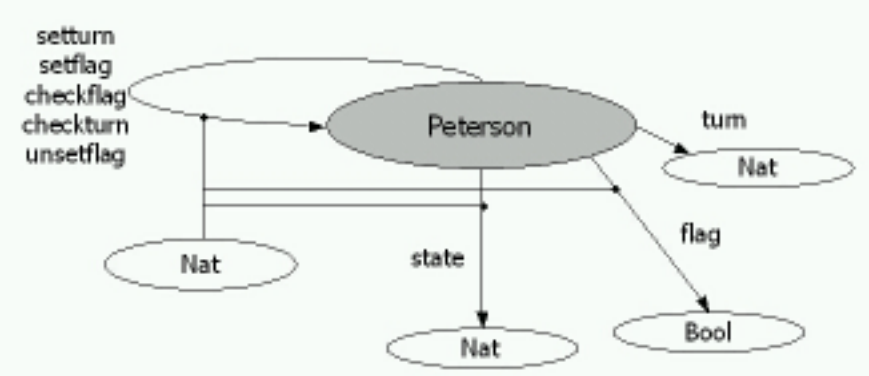

図 5.10: 振舞仕様による Peterson のアルゴリズムのモデル

```
mod! SAFETY{
  pr(BOOL)
  op safety _ : Bool -> Bool {prec: 100}
}

mod* PETERSON2{
  *[ Sys ]*
  pr(NAT + SAFETY)
  bop turn  : Sys     -> Nat .  ** limit * ->  2
  bop state : Sys Nat -> Nat .  ** limit * 2 -> 5
  bop flag  : Sys Nat -> Bool . ** limit * 2 -> *

  bop setturn    : Sys Nat -> Sys . ** limit * 2 -> *
  bop setflag    : Sys Nat -> Sys . ** limit * 2 -> *
  bop checkflag  : Sys Nat -> Sys . ** limit * 2 -> *
  bop checkturn  : Sys Nat -> Sys . ** limit * 2 -> *
  bop unsetflag  : Sys Nat -> Sys . ** limit * 2 -> *

  op  init  : -> Sys

  vars P Q : Nat
  eq  turn(init)   = 0 .
  eq  state(init,P)= 0 .
  eq  flag(init,P) = false .
```

```
  var S : Sys
  ceq state(setflag(S,P),Q)   = 1 if P < 2 and P == Q and state(S,Q) == 0 .
  ceq state(setturn(S,P),Q)   = 2 if P < 2 and P == Q and state(S,Q) == 1 .
  ceq state(checkflag(S,P),Q) = 3 if P < 2 and P == Q and state(S,Q) == 2
                                  and flag(S,s P rem 2) .
  ceq state(checkflag(S,P),Q) = 4 if P < 2 and P == Q and state(S,Q) == 2
                                  and not flag(S,s P rem 2) .
  ceq state(checkturn(S,P),Q) = 2 if P < 2 and P == Q and state(S,Q) == 3
                                  and turn(S) == P .
  ceq state(checkturn(S,P),Q) = 4 if P < 2 and P == Q and state(S,Q) == 3
                                  and not(turn(S) == P) .
  ceq state(unsetflag(S,P),Q) = 0 if P < 2 and P == Q and state(S,Q) == 4 .

  eq turn(setflag(S,P))   = turn(S) .
  ceq turn(setturn(S,P))  = turn(S) if not(P < 2 and state(S,P) == 1) .
  ceq turn(setturn(S,P))  = P       if P < 2 and state(S,P) == 1 .
  eq turn(checkflag(S,P)) = turn(S) .
  eq turn(checkturn(S,P)) = turn(S) .
  eq turn(unsetflag(S,P)) = turn(S) .

  ceq flag(setflag(S,P),Q)   = true      if P < 2 and P == Q and state(S,Q) == 0 .
  ceq flag(setflag(S,P),Q)   = flag(S,Q) if not(P < 2 and P == Q and state(S,Q) == 0).
  eq flag(setturn(S,P),Q)    = flag(S,Q) .
  eq flag(checkflag(S,P),Q)  = flag(S,Q) .
  eq flag(checkturn(S,P),Q)  = flag(S,Q) .
  ceq flag(unsetflag(S,P),Q) = false     if P < 2 and P == Q and state(S,Q) == 4.
  ceq flag(unsetflag(S,P),Q) = flag(S,Q) if not(P < 2 and P == Q and state(S,Q) == 4) .

  ceq state(setflag(S,P),Q)   = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 0) .
  ceq state(setturn(S,P),Q)   = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 1) .
  ceq state(checkflag(S,P),Q) = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 2) .
  ceq state(checkturn(S,P),Q) = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 3) .
  ceq state(unsetflag(S,P),Q) = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 4) .

  -- safety property
  eq safety (state(S,P) == 3 and state(S,Q) == 3) implies P == Q = true .
}
```

図 5.11: Peterson のアルゴリズムに対する振舞仕様

観測演算　この仕様には以下の様に turn, state, flag の 3 つの観測演算がある．

表 5.2: 振舞仕様中の観測演算

観測演算名	意味
turn	アルゴリズム中の turn に対応する．
flag	アルゴリズム中の flag に対応する．
state	アルゴリズム中のプログラムカウンタ値に対応する．

アルゴリズム中では各命令の実行順序は上から下と暗黙的に決められていた．振舞仕様

ではそのような暗黙の仮定をおくことはできない．ここでは各プロセスのプログラムカウンタにあたる観測演算 state を追加する．

操作演算　この仕様には以下の様に５つの操作演算があり，それぞれが 1 つのアトミックな処理に対応する．

- setturn は，プログラムカウンタが 0 の時に限り turn の値を自分のプロセス ID にセットし，カウンタを 1 にセットする．
- setflag は，プログラムカウンタが 1 の時に限り自分のフラグを立て，カウンタを 2 セットにする．
- checkflag は，プログラムカウンタが 2 の時に限り他プロセスのフラグが立っているか調べ，フラグが立っている時はカウンタを 3 にセットし，フラグが立っていないときは 4 似セットする．
- checkturn は，プログラムカウンタが 3 の時に限り turn の値が自分のプロセス ID と等しいか調べ，等しい時はカウンタを 4 にセットし，等しくない時はカウンタを 2 にセットする．
- unsetflag は，プログラムカウンタが 4 の時に限り自分のフラグを下げ，カウンタの値を 0 にセットする．

初期値　この仕様では初期値として定数 init が定義され，以下の 3 つの等式により各プロセスの初期値が設定される．

- 等式 eq turn(init) = 0 . より，turn の初期値は 0 である．
- 等式 eq state(init,P)= 0 . より，全てのプロセスのプログラムカウンタの初期値は 0 である．変数 P について何も条件が無いため，任意の自然数 P に対して state の初期値が 0 に設定される．
- 等式 eq flag(init,P) = false . より，全てのプロセスのフラグの初期値は false である．変数 P について何も条件が無いため，任意の自然数 P に対して flag の初期値が false に設定される．

安全性　この仕様では SMV で行うモデル検査のためにに安全性を記述している．

```
eq safety not(state(S,0) == 4 and state(S,1) == 4) = true .
```

安全性は，“プロセス 0 とプロセス 1 が同時に状態 4 にいることは無いこと” であると記述されている．これはプロセスが状態 4 に移ってから unsetflag を行って状態 0 に遷移する前までの間が Critical Section であり，Critical Section に 2 つのプロセスが同時に存在しないということを記述するとこのようになる．

以上が Peterson のアルゴリズムに対する振舞仕様の説明である．次にこの振舞仕様を SMV 仕様へと変換する．

Peterson のアルゴリズムに対する SMV 仕様

図5.11の振舞仕様を CafeOBJ 仕様変換器 C-TRAS に与え，出力として得られる SMV 仕様を図5.12に示す．

```
MODULE main

VAR
  flag : array 0 .. 1 of boolean;
  state : array 0 .. 1 of 0 .. 4;
  turn : 0 .. 1;
  _nat0 : 0 .. 1;
  __action : {setturn, setflag, checkflag, checkturn, unsetflag};

ASSIGN
  init(turn) :=
  init(state[0]) := 0;
  init(state[1]) := 0;
  init(flag[0]) := FALSE;
  init(flag[1]) := FALSE;

  next(turn) :=
    case
      __action = unsetflag  : turn;
      __action = checkturn  : turn;
      __action = checkflag  : turn;
      __action = setturn  & _nat0 < 2 & state[_nat0] = 1 : _nat0;
      __action = setturn  & !(_nat0 < 2 & state[_nat0] = 1) : turn;
      __action = setflag  : turn;
    esac;

  next(flag[0]) :=
    case
      __action = unsetflag  & !(_nat0 < 2 & _nat0 = 0 & state[0] = 4) : flag[0];
      __action = unsetflag  & _nat0 < 2 & _nat0 = 0 & state[0] = 4 : FALSE;
      __action = checkturn  : flag[0];
      __action = checkflag  : flag[0];
      __action = setturn  : flag[0];
      __action = setflag  & !(_nat0 < 2 & _nat0 = 0 & state[0] = 0) : flag[0];
      __action = setflag  & _nat0 < 2 & _nat0 = 0 & state[0] = 0 : TRUE;
    esac;

  next(flag[1]) :=
    case
      __action = unsetflag  & !(_nat0 < 2 & _nat0 = 1 & state[1] = 4) : flag[1];
      __action = unsetflag  & _nat0 < 2 & _nat0 = 1 & state[1] = 4 : FALSE;
      __action = checkturn  : flag[1];
      __action = checkflag  : flag[1];
```

```
      __action = setturn  : flag[1];
      __action = setflag  & !(_nat0 < 2 & _nat0 = 1 & state[1] = 0) : flag[1];
      __action = setflag  & _nat0 < 2 & _nat0 = 1 & state[1] = 0 : TRUE;
    esac;

  next(state[0]) :=
    case
      __action = unsetflag  & !(_nat0 < 2 & _nat0 = 0 & state[0] = 4) : state[0];
      __action = checkturn  & !(_nat0 < 2 & _nat0 = 0 & state[0] = 3) : state[0];
      __action = checkflag  & !(_nat0 < 2 & _nat0 = 0 & state[0] = 2) : state[0];
      __action = setturn  & !(_nat0 < 2 & _nat0 = 0 & state[0] = 1)   : state[0];
      __action = setflag  & !(_nat0 < 2 & _nat0 = 0 & state[0] = 0)   : state[0];
      __action = unsetflag  & _nat0 < 2 & _nat0 = 0 & state[0] = 4    : 0;
      __action = checkturn  & _nat0 < 2 & _nat0 = 0 & state[0] = 3 &
                                                      !turn = _nat0 : 4;
      __action = checkturn  & _nat0 < 2 & _nat0 = 0 & state[0] = 3 &
                                                       turn = _nat0 : 2;
      __action = checkflag  & _nat0 < 2 & _nat0 = 0 & state[0] = 2 &
                                             !flag[_nat0 + 1 mod 2] : 4;
      __action = checkflag  & _nat0 < 2 & _nat0 = 0 & state[0] = 2 &
                                              flag[_nat0 + 1 mod 2] : 3;
      __action = setturn  & _nat0 < 2 & _nat0 = 0 & state[0] = 1      : 2;
      __action = setflag  & _nat0 < 2 & _nat0 = 0 & state[0] = 0      : 1;
    esac;

  next(state[1]) :=
    case
      __action = unsetflag  & !(_nat0 < 2 & _nat0 = 1 & state[1] = 4) : state[1];
      __action = checkturn  & !(_nat0 < 2 & _nat0 = 1 & state[1] = 3) : state[1];
      __action = checkflag  & !(_nat0 < 2 & _nat0 = 1 & state[1] = 2) : state[1];
      __action = setturn  & !(_nat0 < 2 & _nat0 = 1 & state[1] = 1)   : state[1];
      __action = setflag  & !(_nat0 < 2 & _nat0 = 1 & state[1] = 0)   : state[1];
      __action = unsetflag  & _nat0 < 2 & _nat0 = 1 & state[1] = 4    : 0;
      __action = checkturn  & _nat0 < 2 & _nat0 = 1 & state[1] = 3 &
                                                      !turn = _nat0 : 4;
      __action = checkturn  & _nat0 < 2 & _nat0 = 1 & state[1] = 3 &
                                                       turn = _nat0 : 2;
      __action = checkflag  & _nat0 < 2 & _nat0 = 1 & state[1] = 2 &
                                             !flag[_nat0 + 1 mod 2] : 4;
      __action = checkflag  & _nat0 < 2 & _nat0 = 1 & state[1] = 2 &
                                              flag[_nat0 + 1 mod 2] : 3;
      __action = setturn  & _nat0 < 2 & _nat0 = 1 & state[1] = 1      : 2;
      __action = setflag  & _nat0 < 2 & _nat0 = 1 & state[1] = 0      : 1;
    esac;

  init(_nat0) := {0, 1};
  next(_nat0) := {0, 1};
  init(__action) := {setturn, setflag, checkflag, checkturn, unsetflag};
  next(__action) := {setturn, setflag, checkflag, checkturn, unsetflag};

SPEC AG ((state[0] = 3 & state[0] = 3) -> (0 = 0)) = TRUE
SPEC AG ((state[0] = 3 & state[1] = 3) -> (0 = 1)) = TRUE
SPEC AG ((state[1] = 3 & state[0] = 3) -> (1 = 0)) = TRUE
SPEC AG ((state[1] = 3 & state[1] = 3) -> (1 = 1)) = TRUE
```

図 5.12: Peterson のアルゴリズムに対する SMV 仕様

まず MODULE Sys は，振舞仕様の隠蔽ソート Sys から名前を取っている．MODULE

Sys には VAR 以下 5 つの変数があり，次のような対応関係がある．

- turn は振舞仕様中の turn に対応する．制限の情報により 0 から 1 までの 2 個となる．
- flag は振舞仕様中の flag に対応する．flag はアリティに Nat を取るので配列であり，制限の情報から配列のサイズは 0 から 1 までの 2 つとなる．
- state は振舞仕様中の state に対応する．state はアリティに Nat を取るので配列であり，制限の情報から配列のサイズは 0 から 1 までの 2 つとなる．
- _nat0 は振舞仕様中の各操作演算がアリティに取る Nat に対応する変数である．各操作演算の制限情報から_nat0 がとる値は 0 から 1 までの 2 個となる．この値は実行中のプロセス番号に相当する．
- __action は振舞仕様中の各操作演算に対応する変数である．この値が変化することによって，実行する操作演算が変わる．

各変数の値の変更ルールは ASSIGN 以下に定義されている．turn, flag, state については振舞仕様の等式に対応した変更ルールが生成されている．_nat0, __action については，非決定的に全ての値から選択されるように生成されている．

次にこの仕様に対してモデル検査を行う．

Peterson のアルゴリズムに対するモデル検査

安全性 AG !(a.state[0]=4 & a.state[1]=4) についてモデル検査を行った．検査の結果は true であり，これによりプロセス数 2 まででは，安全性を満たさないような実行経路は存在しないことが分かった．

Peterson のアルゴリズムを任意の個数に拡張することは可能であるが，その際は図5.8のような簡潔さは失われる．他のプロセスがフラグを立てているかどうかを調べる処理がアトミックな処理ではなくなり，また turn による優先順位の決定方法にも変更が必要となる．この問題はプロセス数は 2 であるので有限状態に収まり，制限を与えることなくモデル検査が実行できる．モデル検査から得られた結果 true は，Peterson のアルゴリズムは相互排他を保存することを証明するものである．

5.1.4 Peterson のアルゴリズム (変更版)

次に Peterson のアルゴリズムに変更を加えた以下の仕様に対して同様に検査を行う．この仕様は，Peterson のアルゴリズムの 1・2 番目の動作を入れ替えたものとなっている．

```
0: turn := i
1: flag[i] := true
    while(
2:    flag[~i] and
3:      turn = i)
  <Critical Section>
4:  flag[i] := false

   ただし
          i : {0,1}
          flag[i] : {true, false}
          ~0 = 1, ~1 = 0
   とする.
```

図 5.13: Peterson のアルゴリズム (変更版)

Peterson のアルゴリズム (変更版) に対する振舞仕様

図5.13のプログラムリストに対応する振舞仕様は図5.14の様になる．図5.14は，元の仕様と等式の部分のみ異なるので，等式のみを示す．また前回同様，Critical Section を抜けたプロセスは繰り返し共有資源の獲得を試みる．

```
vars P Q : Nat
eq  turn(init)   = 0 .
eq  state(init,P)= 0 .
eq  flag(init,P) = false .

var S : Sys
ceq state(setturn(S,P),Q)   = 1 if P < 2 and P == Q and state(S,Q) == 0 .
ceq state(setflag(S,P),Q)   = 2 if P < 2 and P == Q and state(S,Q) == 1 .
ceq state(checkflag(S,P),Q) = 3 if P < 2 and P == Q and state(S,Q) == 2
                                    and flag(S,s P rem 2) .
ceq state(checkflag(S,P),Q) = 4 if P < 2 and P == Q and state(S,Q) == 2
                                    and not flag(S,s P rem 2) .
ceq state(checkturn(S,P),Q) = 2 if P < 2 and P == Q and state(S,Q) == 3
                                    and turn(S) == P .
ceq state(checkturn(S,P),Q) = 4 if P < 2 and P == Q and state(S,Q) == 3
                                    and not(turn(S) == P) .
ceq state(unsetflag(S,P),Q) = 0 if P < 2 and P == Q and state(S,Q) == 4 .

ceq turn(setturn(S,P))  = P       if P < 2 and state(S,P) == 0 .
ceq turn(setturn(S,P))  = turn(S) if not(P < 2 and state(S,P) == 0) .
eq turn(setflag(S,P))   = turn(S) .
eq turn(checkflag(S,P)) = turn(S) .
eq turn(checkturn(S,P)) = turn(S) .
eq turn(unsetflag(S,P)) = turn(S) .

eq flag(setturn(S,P),Q)    = flag(S,Q) .
ceq flag(setflag(S,P),Q)   = true      if P < 2 and P == Q and state(S,Q) == 1 .
ceq flag(setflag(S,P),Q)   = flag(S,Q) if not(P < 2 and P == Q and state(S,Q) == 1).
eq flag(checkflag(S,P),Q)  = flag(S,Q) .
eq flag(checkturn(S,P),Q)  = flag(S,Q) .
ceq flag(unsetflag(S,P),Q) = false     if P < 2 and P == Q and state(S,Q) == 4 .
ceq flag(unsetflag(S,P),Q) = flag(S,Q) if not(P < 2 and P == Q and state(S,Q) == 4) .
```

```
ceq state(setturn(S,P),Q)   = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 0) .
ceq state(setflag(S,P),Q)   = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 1) .
ceq state(checkflag(S,P),Q) = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 2) .
ceq state(checkturn(S,P),Q) = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 3) .
ceq state(unsetflag(S,P),Q) = state(S,Q) if not(P < 2 and P == Q and state(S,Q) == 4) .
```

図 5.14: Peterson のアルゴリズム (変更版) に対する振舞仕様 (等式部分)

図5.11の振舞仕様とは，setturn と setflag の実行順序のみが異なる．それ以外については等しい．

Peterson のアルゴリズム (変更版) に対するモデル検査の結果

図5.14の振舞仕様を，C-TRAS により SMV 仕様に変換しモデル検査を行った．その結果，表5.3に示す反例が得られた．表5.3の各ステップでの state, flag, turn の値は，振舞仕様での各ステップの観測演算結果に一致する．しかし__action, _nat0 の値は，次に実行される操作演算とその引数であるので注意されたい．

表 5.3: Peterson のアルゴリズム (変更版) に対するモデル検査結果の反例

	1	2	3	4	5	6	7	8
__action	setturn	setturn	setflag	checkflag	setflag	checkflag	checkturn	checkflag
_nat0	1	0	0	0	1	1	1	0
flag[0]	false	false	false	true	true	true	true	true
flag[1]	false	false	false	false	false	true	true	true
state[0]	0	0	1	2	4	4	4	4
state[1]	0	1	1	1	1	2	3	4
turn	0	1	0	0	0	0	0	0

表5.3は 1 の列が初期状態を表し，$1 \to 2 \to \cdots \to 8$ と状態機械は遷移する．最後の 8 の列では a.state[0] と a.state[1] の値が共に 4 であることから安全性を満たさず，安全性に対する反例となっている．SMV によって返される反例は，反例の中で最も少ない遷移回数で得られる反例である．

この反例が正しいかどうか，CafeOBJ で検証する．図5.15に SMV で得られた反例に対応する検証コードを記述した．初期状態から初め 8 の状態までの遷移が存在すれば，左列全部と右列の 3 段落目までの red 文 は全て true を返す．また 8 の状態が安全性を満たさ

```
open PETERSON2
red flag(init,0) == false .
red flag(init,1) == false .
red state(init,0) == 0 .
red state(init,1) == 0 .
red turn(init) == 0 .
close .

open PETERSON2
red flag(setturn(init,1),0) == false .
red flag(setturn(init,1),1) == false .
red state(setturn(init,1),0) == 0 .
red state(setturn(init,1),1) == 1 .
red turn(setturn(init,1)) == 1 .
close .

open PETERSON2
op a2 : -> Sys .
eq flag(a2,0)  = false .
eq flag(a2,1)  = false .
eq state(a2,0) = 0 .
eq state(a2,1) = 1 .
eq turn(a2)    = 1 .
red flag(setturn(a2,0),0) == false .
red flag(setturn(a2,0),1) == false .
red state(setturn(a2,0),0) == 1 .
red state(setturn(a2,0),1) == 1 .
red turn(setturn(a2,0)) == 0 .
close .

open PETERSON2
op a3 : -> Sys .
eq flag(a3,0)  = false .
eq flag(a3,1)  = false .
eq state(a3,0) = 1 .
eq state(a3,1) = 1 .
eq turn(a3)    = 0 .
red flag(setflag(a3,0),0) == true .
red flag(setflag(a3,0),1) == false .
red state(setflag(a3,0),0) == 2 .
red state(setflag(a3,0),1) == 1 .
red turn(setflag(a3,0)) == 0 .
close .

open PETERSON2
op a4 : -> Sys .
eq flag(a4,0)  = true .
eq flag(a4,1)  = false .
eq state(a4,0) = 2 .
eq state(a4,1) = 1 .
eq turn(a4)    = 0 .
red flag(checkflag(a4,0),0)  == true .
red flag(checkflag(a4,0),1)  == false .
red state(checkflag(a4,0),0) == 4 .
red state(checkflag(a4,0),1) == 1 .
red turn(checkflag(a4,0))    == 0 .
close .
```

```
open PETERSON2
op a5 : -> Sys .
eq flag(a5,0)  = true .
eq flag(a5,1)  = false .
eq state(a5,0) = 4 .
eq state(a5,1) = 1 .
eq turn(a5)    = 0 .
red flag(setflag(a5,1),0) == true .
red flag(setflag(a5,1),1) == true .
red state(setflag(a5,1),0) == 4 .
red state(setflag(a5,1),1) == 2 .
red turn(setflag(a5,1)) == 0 .
close .

open PETERSON2
op a6 : -> Sys .
eq flag(a6,0)  = true .
eq flag(a6,1)  = true .
eq state(a6,0) = 4 .
eq state(a6,1) = 2 .
eq turn(a6)    = 0 .
red flag(checkflag(a6,1),0) == true .
red flag(checkflag(a6,1),1) == true .
red state(checkflag(a6,1),0) == 4 .
red state(checkflag(a6,1),1) == 3 .
red turn(checkflag(a6,1)) == 0 .
close .

open PETERSON2
op a7 : -> Sys .
eq flag(a7,0)  = true .
eq flag(a7,1)  = true .
eq state(a7,0) = 4 .
eq state(a7,1) = 3 .
eq turn(a7)    = 0 .
red flag(checkturn(a7,1),0) == true .
red flag(checkturn(a7,1),1) == true .
red state(checkturn(a7,1),0) == 4 .
red state(checkturn(a7,1),1) == 4 .
red turn(checkturn(a7,1)) == 0 .
close .

open PETERSON2
op a8 : -> Sys .
eq flag(a8,0)  = true .
eq flag(a8,1)  = true .
eq state(a8,0) = 4 .
eq state(a8,1) = 4 .
eq turn(a8)    = 0 .
red not(state(a8,0) == 4 and
        state(a8,1) == 4)
close .
```

図 5.15: Peterson のアルゴリズム (変更版) の反例に対する CafeOBJ での検証

ないならば，最後の red 文は false を返すように作成されている．CafeOBJ で検証した結果，1 から 8 までの状態遷移が存在し 8 の状態が安全性を満たさないことが確認された．

5.2 例題に対する考察

Load と Store による相互排他システムでは，モデル検査を行うことにより自動で反例を発見することができた．一般に証明論的手法による検証では，何の前提も与えること無しにこのように自動的に示すことは難しい．それが SMV により完全自動に検査し，反例が得られたことは本手法が非常に有効していると言える．またモデル検査によって得られた反例に対し，十分な検討を加えることで仕様を修正した．その結果，モデル検査では反例を容易には見つけることができなかった仕様 Test & Set に基づく仕様へと修正することができた．実行系列に依存しない性質 安全性の検証においても，安全性を満たさない状態 (表5.1の場合は 7 の状態) に到達する原因は，それ以前のある状態であること (表5.1の場合は 5 の状態) がほとんどであり，反例を得ることができる重要性を確認した．

相互排他を保証するアルゴリズムとして，PETERSON のアルゴリズムに対しモデル検査を行った．PETERSON のアルゴリズムは有限状態であるため，SMV により安全性を満たすことが証明された．問題が有限状態である場合は，SMV により証明することで自動で検証でき，本手法により容易に検証ができる場合があることを示した．

仕様の証明がうまくいかない時，仕様・性質・検証法のどこに問題があるのかを知ることは困難である．従来は試行錯誤を繰り返して探しており，検証者の能力に大きく依存していた．仕様・性質に問題がある場合，モデル検査を行う事により反例という形で誤りを得られる場合がある．モデル検査は自動的に検証を進めることができるため検証者の負担にならず，導入も容易である．また反例が得られなかった場合であっても，制限された状態では誤りが無いことは保証される．この事実を基底段階とし，帰納的に全状態空間に対して証明することが可能な場合や，より誤りの発生しやすい点を重点的に検証するための指標になる．全状態空間に対する検証が行われなくとも，制限された状態空間に対する検証で得られる情報は少なくない．また多くの誤りは，無限状態でなくとも非常に少ない状態で現れることがほとんどである．

第 6 章

結論

6.1 まとめ

本研究では，状態遷移機械を記述する振舞仕様に対して，証明論的手法に基づく検証器 CafeOBJ とモデル検査的手法に基づく検証器 SMV を併用した検証法を提案した．振舞仕様に対し SMV によりモデル検査を行うため，仕様変換器 C-TRAS を実装した．

振舞仕様は無限状態を持つ状態遷移機械が記述可能なのに対し，SMV 仕様は有限状態機械のみを扱うことが可能である．このため C-TRAS は，制限という方法により振舞仕様の状態空間を有限化して変換を行う．制限によって有限化した状態遷移機械に対してモデル検査を行い得られる反例は，元の振舞仕様の一部の誤りであり，全てが得られるとは限らない．

また本研究では，C-TRAS によって生成された SMV 仕様に対してモデル検査を行い，反例が得られた時，その反例が元の振舞仕様に対しても反例であることを証明した．これにより，SMV が報告した反例が，本当に反例であるか CafeOBJ によって検証を省略可能にした．

実際に Load と Store に基づく相互排他システム，Test & Set に基づく相互排他システム，Peterson のアルゴリズムに対して実験を行った．振舞仕様をある範囲に制限し，モデル検査を行った．いくつかのモデルにおいて反例が報告されたので，その反例が元の振舞仕様においても反例である事を CafeOBJ によって確認した．証明論的手法による証明ではこれらの反例は自動的に得られないため，振舞仕様に対するモデル検査の有効性を確認した．また Load と Store に基づく相互排他システムでは，SMV によって報告される反例よりその原因を考察し，仕様を修正した．修正した仕様に対しては SMV により反例が報告されなくなり，明らかな誤りを修正することができた．本研究は，振舞仕様に対し SMV により反例を探すことで CafeOBJ による検証を支援する．誤りが存在することだけでな

く反例を示すことができるため，誤りの原因を考察を助け，仕様の修正を容易にする．

無限状態をもつ仕様に対し容易にモデル検査を行うことはできない．抽象化は性質を保存しながら状態を有限化し，モデル検査を行う．抽象化により，モデル検査により無限状態遷移機械に対し証明が可能になる点はすばらしい．しかし，抽象化法が簡単に得られない場合，モデル検査により誤りの発見についても行うことができなかった．本研究では，そのような場合であっても，状態を制限によって有限化することにより，ある程度の場合について網羅的に検査を行えるようにした．モデル検査の利点である反例報告を，早期に利用できることにより検証を支援することができるようになった．

6.2 関連研究

HOL-to-SMV HOL-to-SMV [ZB01] は，証明論的手法の一つである HOL に対する仕様から SMV 仕様に変換するための変換器である．両仕様の表現能力の違いから，本研究と同様に扱うことのできるデータ型に制限を与えているが，組や単純な集合を扱うことができる．しかし等価関係などを計算するために，変数を割当て不必要に状態数を増加させているため状態爆発を起こしやすい．本研究では等価関係の計算に変数を割当てず条件式で行うため，HOL-to-SMV よりも処理効率の良い仕様を生成する．また HOL-to-SMV は，有限状態の HOL 仕様を SMV に変換することを対象としている点でも，制限を用いて有限化を行う本研究と異なる．

Maude LTL Model Checker Maude [mau] は，SRI で実装された代数仕様記述言語である．LTL Model Checker [EMS02] は，Maude で記述された代数仕様言語に対し LTL に基づくモデル検査を行う．任意の抽象データ型を扱う仕様に対してモデル検査を行う事ができる．本研究が振舞仕様をモデル検査の対称とするのに対し，LTL Model Checker は隠蔽ソートを使用しない可視ソート上の仕様を対象とする．また，本研究が制限により有限化した仕様に対し安全性の検査を行うのに対し， Maude LTL Model Checker は到達可能状態が有限であれば，モデル検査による検証を行うことができる．また検証する性質が安全性であれば，無限であっても時間の許す限り検証を行う．

SAL SAL(Symbolic Analysis Laboratory) [Sha00, BGL+00] は検証器のための中間言語である．SAL 言語に対し抽象化・スライシングを適用することができ，問題を単純化させることができる．また SAL から PVS, SMV への変換器を持ち，それぞれ検証器によって検証することができる．無限状態を持つ仕様に対しモデル検査を行う場合，抽象化によって有限状態機械に変換してから行う．SAL の有限状態遷移機械は SMV と自然な対応関係を取ることができ，全て変換できる．

本研究では，検証する対象として無限状態遷移機械を用いた．この状態繊維機械を有限状態にするため制限を用いた．

6.3 今後の課題

現在の変換器はいくつかの機能制限や未実装の機能がある．これらの機能を実装し，任意の振舞仕様に対しモデル検査を可能にすることが挙げられる．

- ユーザ定義のデータモジュールを使用可能にすること．
  現在の変換器では，抽象データ型として自然数とブール代数のみを利用可能としていた．問題により適した抽象データ型を使うことにより仕様中に誤りが混入することを未然に防ぎ，理解の容易な仕様を作成することができる．

- 安全性以外のモデル検査．
  現在記述できる性質は安全性に限られている．これは安全性がシステム対して最も重要な性質であるため，安全性の検証から取り組んだ．しかし，CafeOBJ のデータモジュールとして CTL 言語を定義することで，その他の性質も容易に変換が可能である．

- プロジェクションを用いた振舞仕様に対する変換の実装．
  プロジェクションは，振舞使用の再利用や問題の分割に貢献する．1 つの大きな状態遷移機械が，小さな状態遷移機械の合成によって表現される場合に，プロジェクションが用いられる．SMV にもモジュール (状態遷移機械) の再利用をすることができる．データ型の中には既に定義されたモジュールを使用することができるようになっている．プロジェクションと SMV モジュールの階層化の間の関係について対応関係をとり，プロジェクションの利用を可能にする．

- 常時は観測できない状態変数を持つ振舞仕様の変換．
  現在扱うことができる振舞仕様は，観測演算によって得られる情報の組から現在の状態を一意に決定可能であるように設計されている．3.1節で説明した Stack を振舞仕様で記述すると，観測演算 top を実行してもその下に何が詰まれているかは観測できない．このような場合，観測演算で得られる情報を配列で持ち，実行経路によって配列中の読み書きできる場所を移すことにより等価な振舞機械を定義できる．

- 振舞合同を扱った振舞仕様に対するモデル検査．
  現在の振舞仕様は，振舞合同を用いた振舞仕様は想定していない．振舞合同である 2 状態に対し，同じように遷移規則を適用させれば，常に同じ観測結果を返す．先の

Stack のように，観測演算からは常には観測できない情報についても等しいことを示す．振舞合同を扱った仕様に対して，モデル検査を用いた安全性検査が提案されているので，それを参考に実現法を考案する．

次にモデル検査が扱うことできる仕様・検証能力の拡張が挙げられる．モデル検査は全状態空間を網羅的に調べるために扱うことができる仕様は有限に限られていた．

- Compositional Model Checking による検証．
  Compositional Model Checking は有限状態機械を合成して得られる無限状態遷移機械に対するモデル検査法である．有限状態機械を反復的に合成して，合成状態遷移機械の不動点を求め，各合成結果に対するモデル検査を行う．反復的に合成して得られた有限状態機械全てに対しモデル検査で正しいことが証明されれば，任意の合成に対する状態遷移機械の証明を与えたことになる．無限状態を持つ状態遷移機械の証明をモデル検査で行う．今回例題に用いた相互排他問題は，この手法によりモデル検査でも自動で証明が可能である．

- 抽象化．
  無限状態をモデル検査で扱う方法として，現在最も活発に研究されているのが抽象化である．抽象化によって有限状態機械へと変換し，仕様の正しさを証明することができる．抽象化は，モデル検査に限らず検証において重要な概念である．ある種の問題に対する抽象化を変換器に実装することで，扱うことができる問題を大きく増やすことができる．

最後に，検証器を拡張するメタ言語の必要性が挙げられる．本研究では，C-TRAS を実装するために振舞仕様のパーズ処理など全ての処理を実装する必要があった．今後，検証器は抽象化など様々な機能を追加していく必要がある．これらの要求に応えるために，検証器は自身の能力を拡張する能力が必要である．

SRI で実装された代数仕様記述言語 Maude はリフレクションを用いて実装されている．メタな機能により構文の拡張や簡約処理の変更など柔軟に対応できる．Isabble/HOL は ML を内部言語に持っており，ML で証明規則を記述するだけでなく，処理系自体の能力を追加することができる．HOL2SMV は ML により，Voss は fl という ML-like な言語で記述されたモデル検査を行うためのシステムであり，拡張機能を利用して実装されている．

検証器の能力を拡張する上で ML がどのように有効に機能しているかについて調べ，検証器の拡張に適した内部言語について調査する．

謝辞

本研究を進めるにあたり，指導して頂いた二木厚吉教授に深く感謝致します．また，有益な助言をして頂いた緒方和博氏，中村正樹助手，天野憲樹助手に御礼を申し上げます．最後に，公私にわたり付き合って頂いた言語設計学講座の諸氏に感謝致します．

参考文献

[AJS99] Mark D. Aagaard, Robert B. Jones, and Carl-Johan H. Seger. Lifted-fl: A pragmatic implementation of combined model checking and theorem proving. In *Theorem Proving in Higher-Order Logics*. Springer-Verlag, September 1999.

[BGL+00] Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Mu noz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pp. 187–196, Hampton, VA, jun 2000. NASA Langley Research Center.

[cad] Cadence SMV. URL `http://www-2.cs.cmu.edu/~modelcheck/smv.html`.

[caf] CafeOBJ. URL `http://www.ldl.jaist.ac.jp/cafeobj/`.

[CCGR98] A. Cimatti, E. Clarke, F. Giunchiglia, and M. Roveri. Nusmv: a reimplementation of smv. Technical report, IRST, Trento, Italy, 1998.

[CGL94] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, Vol. 16, No. 5, pp. 1512–1542, September 1994.

[ea01] B. Berard M. Bidoit A. Finkel F. Laroussinie et al. Systems and Software Verification : Model-Checking Techniques and Tools. Springer-Verlag, 2001.

[ED99] Orna Grunmerg Edmund M. Clarke, Jr. and Doron A. Peled. *Model Checking*. The MIT Press, London, England, Dec. 1999.

[EMS02] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. The Maude LTL model checker. In Fabio Gadducci and Ugo Montanari, editors, *Fourth Workshop on Rewriting Logic and its Applications, WRLA '02*, Vol. 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, September 2002.

[EOD93] E.Clarke, O. Grumberg, and D. Long. Verification Tools for Finite State Concurrent Systems. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *A Decade of Concurrency-Reflections and Perspectives*, Vol. 803, pp. 124–175, Noordwijkerhout, Netherlands, 1993. Springer-Verlag.

[Gor00] Michael J. C. Gordon. Reachability programming in hol98 using bdds. In *Theorem Proving and Higher Order Logics*. Springer-Verlag, August 2000.

[Gor02] Michael J. C. Gordon. *HolBddLib Version 2 Documentation*, March 2002.

[JS93] Jeffrey Joyce and Carl Seger. The HOL-Voss system: Model-checking inside a General Purpose Theorem-Prover. In *International Workshop on the HOL theorem proving system and its applications*, No. 780 in Lecture Notes in Computer Science. Springer-Verlag, 1993.

[KP99] Yonit Kesten and Amir Pnueli. Verifying liveness by augmented abstraction. In *CSL*, pp. 141–156, 1999.

[mau] Maude. URL `http://maude.cs.uiuc.edu/`.

[NPW] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL A Proof Assistance for Higher-Order Logics*. No. 2283 in Lecture Notes in Computer Science. Springer-Verlag.

[nus] NuSMV. URL `http://nusmv.irst.itc.it/`.

[RK98] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*. AMAST Series. World Scientific, 1998.

[Sha00] Natarajan Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR'00: Concurrency Theory*, No. 1877 in Lecture Notes in Computer Science, pp. 1–16, State College, PA, aug 2000. Springer-Verlag.

[ZB01] Dan Zhou and Paul E. Black. Translating HOL to Specifications for the Model Checker SMV. In *Theorem Proving in Higher Order Logics supplemental proceedings*, pp. 400–415, September 2001.