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

Title	モデル検査における誤り原因の特定に関する研究
Author(s)	小川，直哉
Citation	
Issue Date	2014-03
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/12057
Rights	
Description	Supervisor:青木利晃　准教授，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

修 士 論 文

モデル検査における誤り原因の特定に関する研究

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

小川 直哉

2014 年 3 月

修 士 論 文

モデル検査における誤り原因の特定に関する研究

指導教員　青木利晃 准教授

審査委員主査　青木利晃 准教授
審査委員　鈴木正人 准教授
審査委員　緒方和博 准教授

北陸先端科学技術大学院大学
情報科学研究科情報科学専攻

1210016 小川 直哉

提出年月: 2014 年 2 月

Copyright © 2014 by Ogawa Naoya

概 要

本稿では、モデル検査ツール Spin を対象に、検証の誤りの原因を特定する手法を提案する。モデル検査の誤りには、検査対象の誤り、検査対象のモデル化の誤り、性質の与え方の誤りの３つがある。提案する手法では、出力された反例を正標本集合、負標本集合と比較することで誤りの原因を特定する。そのためにまず正標本集合、負標本集合、及び標本集合による特定手法を形式的に定義した。実験は典型的な事例を対象に行なった。

目 次

第 1 章　はじめに **1**

第 2 章　背景 **2**
2.1　モデル検査と Spin 2
2.2　研究の動機 2

第 3 章　集合論に基づく誤り特定手法の形式化 **5**
3.1　反例と真の反例 5
3.2　モデルの正確性 6
3.3　性質の健全性 6
3.4　反例に関する命題 6
3.5　標本集合 7
3.6　標本集合による特定 8

第 4 章　負標本集合による誤り特定 **9**
4.1　構文定義 9
4.2　判定方法 10
4.3　実験 14
4.3.1　計画 14
4.3.2　実験内容 14
4.3.3　評価 24
4.3.4　考察 25

第 5 章　正標本集合による誤り特定の実験 **28**
5.1　実験 28
5.1.1　計画 28
5.1.2　実験内容 28
5.1.3　考察 37

第 6 章　まとめ **39**

第 7 章　今後の課題 **40**

第1章　はじめに

形式検証はソフトウェアの信頼性向上のために用いられる技術の 1 つである。形式検証の 1 つにモデル検査がある。モデル検査では検査対象を有限状態遷移系にモデル化し、網羅的探索技法を用いることで与えられた性質を満足するかどうか検査する手法である。しかし、検査対象のモデル化自体に誤りがあると検査対象の不具合を発見することは出来ない。モデル化に誤りがある場合、誤りの原因は振る舞いのモデル化、または性質の与え方にあるがその発見はモデルが複雑になると難しくなる。

本研究の目的はモデル検査ツール Spin を対象に、検証の誤りの原因を特定する手法を提案することである。モデル検査はまず検査対象の振る舞いを仕様記述言語[1] でモデル化し、次に検査対象が満たすべき性質を時相論理式などで与える。モデル化された検査対象の状態数は有限であるので、すべての状態を網羅的に探索することで与えられた性質を満足するかどうか検査することが出来る。ここで検査対象のモデル化や性質の与え方自体に誤りがあると検査対象の不具合を発見することが出来ないという問題がある。本来のものと異なるモデルを探索し、そこで成り立つ性質を調べたとしてもその検証は無意味だからである。そこで本研究ではその原因が検査対象のモデル化にあるのか性質の与え方に問題があるのか特定する手法を提案する。

本研究の特色は検査対象のモデル化が十分されていなくとも誤り原因の特定を見込めることである。本研究では標本集合を用いた検証の誤りの原因を特定する手法を提案する。本研究における標本集合とは検査対象の性質を満たす振る舞いだと確信できるもの（正標本集合）と検査対象の振る舞いではないと確信できるもの（負標本集合）を指し具体的な検査対象の実行例である。誤り原因の特定は反例と標本集合を比較することで行なう。モデル検査の結果、もし反例が正標本集合に含まれているのならその反例は偽反例ということであり性質の与え方が誤っていたと言える。一方、負標本集合に含まれているなら検査対象のモデル化が誤っていたと言える。しかし、反例が標本集合に含まれていない場合も存在するし、正標本集合に基づいたとしても検査対象のモデル化にも誤りがある場合も存在する。そこで、これらの問題点を踏まえて幾つかの事例研究を行い、提案手法が適用できる条件、標本集合の位置づけを整理し、提案手法の評価・考察する。

[1]本研究では PROMELA を用いた。

第2章　背景

2.1　モデル検査とSpin

形式検証はソフトウェアの信頼性向上のために用いられる技術の1つである。モデル検査とは形式検証の1つである。モデル検査はまず検査対象の振る舞いを仕様記述言語、検査対象が満たすべき性質を時相論理式などを用いてモデル化する。モデル化された検査対象の状態数は有限であるので、すべての状態を網羅的に探索することで与えられた性質を満足するかどうか機械的に検査することが出来る。

Spin [1] は通信プロトコルの検証を目的に、G.J.Holzmann らによって開発されたモデル検査のためのツールであり、実システムとして使われるソフトウェアのモデル検査を次のような手順で行なう。まず、システムの振る舞いを表すモデルを PROMELA [2] と呼ばれる言語で記述する。続いて、その振る舞いの性質を論理式で指定することでシステムの要求定義を表す。最後に記述した振る舞いの中でその性質をモデルが満たしているか自動的に検査する。もしモデルが性質を満たさない場合はその反例、すなわち期待する性質に反する実行例を示す。PROMELA の基本文法はプログラミング言語である C 言語によく似ている。

2.2　研究の動機

研究の動機は、モデル検査の性質の与え方に難しさを感じたからである。モデル検査の誤りの原因は必ずしも検査対象にあるとは言えない。検査対象の誤りを検出するには、検査対象を忠実かつ簡潔に表現しており、正当性が検証可能なモデルを構築し、さらに時相論理式などで検証対象が満たす性質を表現する必要がある。検査対象のモデル化や性質の与え方が誤っていた場合、一般には、性質の与え方の方が相対的に検査対象のモデル化より単純であるため検査対象のモデル化が誤っていると結論づけることが多い。しかし、性質の意味を勘違いしていることも少なくはない。このようにモデル検査の誤りには「検査対象の誤り」、「検査対象のモデル化の誤り」、「性質の与え方の誤り」の３種類がある。

具体的に、信号機を検査対象にして Spin で検査する例を用いてこれらの誤りを示す。「検査対象の誤り」は、時間帯で信号の振る舞いを変えるようにしたつもりが出来ていない、信号が一向に変化しない状況になることがある、などが挙げられる。「検査対象のモ

[1] Simple Promela INteroreter の頭文字が由来である。
[2] PROcess MEta LAnguage の頭文字が由来である。

デル化の誤り」は、扱う変数の誤りが挙げられる。リスト 2.1 は十字交差点での時差式信号機のモデルの一部分をのせたものである。7 行目で、信号 A に出すはずのメッセージを信号 B に送っている。

リスト 2.1: 振る舞いのモデル化の誤りの例

```
1  proctype ctr(chan to_sgnA, to_sgnB, to_sgnC, to_sgnD)
2  {
3      do
4      ::((stateA == blue) && (stateB == red)
5         && (stateC == blue) && (stateD == red)) ->
6          if
7          ::to_sgnB!msg(yellow);
8            if
9            ::to_sgnA!msg(red); to_sgnC!msg(yellow); to_sgnC!msg(red)
10           ::to_sgnC!msg(yellow);
11             if
12             ::to_sgnA!msg(red); to_sgnC!msg(red)
13             ::to_sgnC!msg(red); to_sgnA!msg(red)
14             fi
15           fi
16         .
17         .
18         .
19         fi;
20         ((stateA == red) && (stateC == red)) ->
21         to_sgnB!msg(blue); to_sgnD!msg(blue);
22         .
23         .
24         .
25         ((stateB == red) && (stateD == red)) ->
26         to_sgnA!msg(blue); to_sgnC!msg(blue);
27     od
28 }
```

「性質の与え方の誤り」は、青から黄への変化を保証する性質を、誤った時相論理式で記述していることが挙げられる。以下に LTL 式で誤った記述と正しい記述を示した。信号が青になったとき黄になるまでは青の状態が続くことで表現できる。'[]' は後ろに続く式が常に成り立つことを意味する。

誤った記述

```
r → [](p U q)
  r:(stateA == blue),  p:(stateA == blue),  q:(stateA == yellow)
```

正しい記述

```
[](r → (p U q))
  r:(stateA == blue),  p:(stateA == blue),  q:(stateA == yellow)
```

第3章　集合論に基づく誤り特定手法の形式化

モデル検査では、検査対象のモデル (M) と、満たすべき性質 (P) を記述する。ここで、反例が出力された場合、$M \not\models P$ であることは言えるが、M が誤っているのか、P が誤っているのか判断がつかない場合がある。そこで、出力された反例に基づいて、M が誤っているのか、P が誤っているのか特定する手法を提案するのが本研究である。

2.2 で信号機の例をもとにモデル検査の誤りを 3 種類挙げた。検査対象のモデル化及び、性質の与え方に誤りがある場合、モデル検査で反例が得られたからといって検査対象に誤りがあるとは言えない。M や P は人が作ったものであり、検査対象に問題がなかったとしても、人為的な誤りによってモデルが性質を満たさないと判断されてしまうからである。そこでまず、反例が検査対象の誤りを指すものなのか、そうでないのか区別する必要が出てくる。そのために、モデル検査で扱う**モデル**M と**性質**P とは別に、モデル作成者の意図する**真のモデル**$\tilde{M}$ と**真の性質**$\tilde{P}$ を導入する。すなわち、モデル作成者は自分の思い描く $\tilde{M}$ から M を作成し、$\tilde{P}$ から P を作成することになる。これら $\tilde{M}$, $\tilde{P}$, M, P は振る舞いの集合を表す。

モデル検査は $M \not\models P$ である場合、反例を出力する。反例は M が P を満たさないことを示す実行例の 1 つである。モデル検査で得られた、この反例が検査対象そのものの誤りを指している場合には、反例は $\tilde{M}$ が $\tilde{P}$ を満たさないことを示す実行例であると言える。反例が検査対象の誤りを指す場合、真の反例と呼ぶことにする。

3.1　反例と真の反例

定義 3.1.1 (反例と真の反例) 振る舞い $e \in B$ が下記を満たすとき、e を**真の反例**と呼ぶ。
$e \in \tilde{M}$ かつ $e \notin \tilde{P}$。
また、振る舞い $e \in B$ が下記を満たすとき、e を**反例**と呼ぶ。
$e \in M$ かつ $e \notin P$。

反例が真の反例（検査対象の誤り）を指すかどうかは、$e \in M$ と $e \notin P$ から $e \in \tilde{M}$ と $e \notin \tilde{P}$ が導ければ良い。そのため、$\tilde{M}$ と M、$\tilde{P}$ と P の関係について述べる。まず $\tilde{M}$ と M について、検査対象は正確にモデル化できていなければならないとした。モデル化した

ものが余計な振る舞いをする可能性があったり、或る振る舞いが起こらないことがある場合、それは別の検査対象をモデル化してしまったことになると考えたからである。一方、$\tilde{P}$ と P については、検査対象が満たす性質を表現できていれば良いので P に必要以上の情報が入っていても構わないとした。以下で $\tilde{M}$ と M、$\tilde{P}$ と P の関係について定義する。

3.2 モデルの正確性

定義 3.2.1 (モデルの正確性) $M = \tilde{M}$ が成立するとき、モデル M は真のモデル $\tilde{M}$ に対して**正確である**と言う。

時間帯で振る舞いが変わる信号機のモデルを書こうとしたとき、作成者の思い描くモデルが書けた場合、モデル M は真のモデル $\tilde{M}$ に対して正確であると言える。実際に、時間帯で振る舞いが変わるように作成できたかはモデル検査で明らかにすべきことであり、作成者はそこで初めて、自分の思い描いたものは誤りだと気づく。

3.3 性質の健全性

定義 3.3.1 (性質の健全性) $\tilde{P} \subseteq P$ が成立するとき、性質 P は真の性質 $\tilde{P}$ に対して**健全である**と言う。

「いつか X が成り立つ」ことを検査したいときに、時相論理式で `<>X` と記述するところを、「常に X が成り立つ」ことを意味する `[]X` と記述してしまった場合、性質 P は真の性質 $\tilde{P}$ に対して健全であると言える。常に成り立つことが言えれば、いつか成り立つことも言えるからである。

$\tilde{M}$ と M、$\tilde{P}$ と P の関係が定義できたので反例に関する命題とその証明を示す。

3.4 反例に関する命題

命題 3.4.1 (反例に関する命題) モデルの正確性と性質の健全性が満たされるとき反例は真の反例である。

(証明) 定義 3.1.1、定義 3.2.1 より

$$e \in M, M = \tilde{M}$$

よって

$$e \in \tilde{M} \qquad ---①$$

定義 3.1.1、定義 3.3.1 より

$e \notin P, \tilde{P} \subseteq P$

よって

$e \notin \tilde{P}$ ---②

①、②、定義 3.1.1 より

e は真の反例となる。 □

モデル検査ではモデル M が性質 P を満たさないとき反例を出力するが、本来、反例として出てきてほしくないもの（偽反例）が出力されることもある。反例が偽反例であるかそうでないかは或る基準（正しさの前提）が存在するので判断できる。そこで、正しいと認める集合 (正標本集合) と、誤っていると認める集合 (負標本集合) を導入する。正標本集合とは検査対象の性質を満たす振る舞いの集合である。負標本集合とは検査対象の振る舞いではないと確信できる振る舞いの集合を表す。これらは標本なので、正標本集合、負標本集合の要素を全て集めてくる必要はない。これらの集合は、全体の振る舞いの一部なので、相対的に確信度が高い．よって、そのような確信を持てる振る舞いを前提として、M と P のどちらが誤っているか判定を試みる。これが正しさの前提となる。

3.5 標本集合

定義 3.5.1 (正標本集合) 振る舞いの集合 $S_p \subseteq B$ が下記を満たすとき、S_p を**正標本集合**と呼ぶ: $S_p \subseteq \tilde{P}$.

十字交差点での時差式信号機の例を考える。左から時計回りに信号 A、信号 B、信号 C、信号 D と割り振る。信号 A の向い側に信号 C、信号 B の向い側に信号 D があることになる。振る舞いは変数の遷移で表すことにする。信号機の場合は、*blue*、*yellow*、*red* の値をとる変数の 4 つ組 (A, B, C, D) になる。今、信号の安全性[1] を検査しようとしたとき $(blue, red, blue, red) \Rightarrow (yellow, red, blue, red) \Rightarrow (red, red, blue, red)$ や $(blue, red, blue, red) \Rightarrow (yellow, red, blue, red) \Rightarrow (yellow, red, yellow, red)$ また、$(blue, red, blue, red) \Rightarrow (blue, red, yellow, red) \Rightarrow (blue, red, red, red)$ などが、正標本集合の要素となる。正標本集合とはこれら、検査対象の性質を満たす振る舞いだと確信できるものの集合である。

定義 3.5.2 (負標本集合) 振る舞いの集合 $S_n \subseteq B$ が下記を満たすとき、S_n を**負標本集合**と呼ぶ: $S_n \cap \tilde{M} = \emptyset$.

先ほどの十字交差点での時差式信号機の例を考える。信号の振る舞いとして明らかにおかしいもの、$(blue, red, blue, red) \Rightarrow (blue, blue, blue, red) \Rightarrow (yellow, blue, blue, red)$ や $(blue, red, blue, red) \Rightarrow (red, red, blue, red)$ などが、負標本集合の要素となる。負標本集合

[1]例えば信号 A が青のとき、隣り合う信号 B も青だと安全とは言えない。

とはこれら、検査対象の振る舞いではないと確信できるものの集合である。

正しさの前提を定義できたので標本集合による特定に関する定理とその証明を示す。検査対象のモデル化の誤りは正標本集合によって特定でき、性質の与え方の誤りは負標本集合によって特定できる。

3.6 標本集合による特定

定理 3.6.1 (正標本集合による誤り原因の特定) 反例 e が正標本集合に含まれる場合、性質の健全性が成立しない。さらに，e は真の反例でない。

(証明) 反例が正標本集合に含まれていること及び、定義 3.5.1 より

$e \in S_p, S_p \subseteq \tilde{P}$

よって

$e \in \tilde{P}$ ---①

定義 3.1.1 より

$e \notin P$ ---②

①、②、定義 3.3.1 より

性質の健全性が成立しない。

さらに、命題 3.4.1 より

e は真の反例ではない。 □

定理 3.6.2 (負標本集合による誤り原因の特定) 反例 e が負標本集合に含まれる場合、モデルの正確性成立しない。さらに、e は真の反例でない。

(証明) 定義 3.5.2 より

$S_n \cap \tilde{M} = \emptyset$

よって

$S_n \subseteq \tilde{M}^c$ ---①

反例が正標本集合に含まれていること及び、① より

$e \in \tilde{M}^c$

よって

$e \notin \tilde{M}$ ---②

定義 3.1.1 より

$e \in M$ ---③

②、③、定義 3.2.1 より

モデルの正確性が成立しない。

さらに、命題 3.4.1 より

e は真の反例ではない。 □

第4章　負標本集合による誤り特定

この章ではまず、有効な負標本集合の書き方について述べる。負標本集合は反例と比較する必要があるため、比較しやすい形で書くことが求められる。次に反例が負標本集合に含まれているか判定する手順を示す。書き方と判定方法は具体例を示す。最後に負標本集合の評価のための実験について述べる。

4.1　構文定義

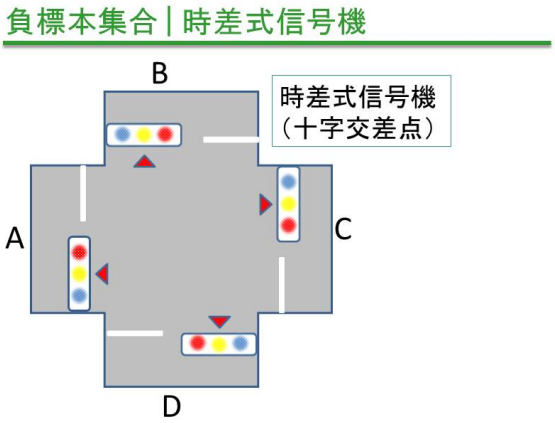
負標本集合とは検査対象の振る舞いではないと確信できるものである。3.5.2 の時差式信号機の例だと $(blue, red, blue, red) \Rightarrow (blue, blue, blue, red) \Rightarrow (yellow, blue, blue, red)$ や $(blue, red, blue, red) \Rightarrow (red, red, blue, red)$ のように書いた。しかし、単にデータの遷移を次々に羅列していくのは大変である。そこで、おかしなデータの遷移が 1 つでも含まれていたらその遷移列は負標本集合の要素であると言えることに着目し、負標本集合の書き方を定義した。これは書き方を決めたものであり、負標本集合の定義とは異なる。

負標本集合を書くために変数の組の遷移を要素とする集合 U を導入する。p を遷移前の変数の組、p' を遷移後の変数の組、c を変数に関する条件として U を以下のように定義する。

$$\boxed{U = \langle p \Rightarrow p' | c \rangle}$$

変数の条件 c に誤りとなる条件を記せば負標本集合は U の要素を含む変数の遷移の集合と意味づけられる。ここで時差式信号機を例に実際に記述してみる。

–



信号は左から時計回りに A,B,C,D とする。振る舞いは、まず信号 A と信号 C が青の状態から共に赤になる。このとき色が変化する順番は問わない。例えば信号 A が先に黄、赤と変化し次に信号 C が黄、赤と変化することもあれば信号 A が黄、信号 C が黄、赤、信号 A が赤と変化することもある。共に赤になったら次は信号 B と D が青になる。起こりえない振る舞いの 1 つとして『隣り合う信号がどちらも赤ではない』ことが挙げられる。「隣り合う」とは、信号 A と「隣り合う」信号は、信号 B と信号 D であるという意味である。この起こりえない振る舞いを集合 U の要素 u_1 を用いて記述する。変数の組は (A, B, C, D) とする。A, B, C, D は各信号を表し、$blue, yellow, red$ の値をとる。

『隣り合う信号がどちらも赤ではない』は次のように書ける。

$$u_1 \equiv (A, B, C, D) \Rightarrow (A', B', C', D') \ \ when \ \ (A' \neq red \wedge B' \neq red) \vee (B' \neq red \wedge C' \neq red) \vee (C' \neq red \wedge D' \neq red) \vee (D' \neq red \wedge A' \neq red)$$

他に起こりえない振る舞いとして『各信号が青⇒黄⇒赤の順に変化しない』ことが挙げられる。同様に集合 U の要素 u_2 を用いて記述していけば、負標本集合の要素を増やすことができる。

4.2 判定方法

モデル検査の反例が負標本集合に含まれるかどうかは反例と集合 U とのマッチングをとることで行なう。集合 U の要素は変数の遷移であるため反例は変数の遷移列である必要がある。そのために変数の遷移が起こるたびに着目する変数の値を出力するようにモデルを修正する。但し不可分実行を行なうときは処理の最後に変数の出力を行なうものとする。

モデルの誤りの判定は以下の手順で行なう。

- 反例 (変数の遷移列) と負標本集合のもとになる集合 U を用意する。
- 反例に集合 U の要素が含まれるような遷移が含まれているかチェックする。

- 含まれていれば『モデルに誤りがある』と判定する。
- 含まれていなければ反例の次の遷移について同様にチェックする。
- どの遷移も集合 U に含まれていないときは判定不可能とする。

集合 U というのは全ての起こりえない振る舞いを考慮しているわけではない。なので手順の最後で判定不可能とする理由は、単に集合 U の書き漏らしによって判定できないという可能性があるからである。

反例が n 個の遷移で出力されたとき、これらの手順を疑似コードで表すと次のようになる。

リスト 4.1: 判定方法の疑似コード

```
int i,j;

for(1<=i<n){
    for(p⇒p' if condition)∈U{
        if(∃σ, σ(p⇒p')=di⇒di+1∧σ(condition)){
            print(di⇒di+1,condition);/*反例の遷移と引っかかった条件を出力*/
            return;
        }
    }
}
/*集合Uの要素が見つからないとき*/
print(''判定できません.'');
exit;
```

疑似コードで σ は変数の割り当てを表す。

先ほどの時差式信号機を例として挙げる。モデルに故意に誤りを混入させて、反例が負標本集合に含まれることを示す。信号機はコントローラを介したメッセーッジ通信で表現した。誤りを混入させたコントローラのモデルと検査する性質は次のようになる。但しモデルは変数の遷移が起こるたびに着目する変数の値を出力するように修正してある。また検査する性質は LTL [1]式を用いた。

リスト 4.2: 時差式信号機

```
proctype ctr(chan to_sgnA, to_sgnB, to_sgnC, to_sgnD)
{
    do
    ::((stateA == blue) && (stateB == red)
       && (stateC == blue) && (stateD == red)) ->
        if
        ::to_sgnA!msg(yellow);
          if
          ::to_sgnA!msg(red); to_sgnC!msg(yellow); to_sgnC!msg(red)
```

[1]Spin での検証に用いられる形式論理である線形時間論理 (LTL:Linear Temporal Logic)

```
            :: to_sgnC!msg(yellow);
              if
              :: to_sgnA!msg(red); to_sgnC!msg(red)
              :: to_sgnC!msg(red); to_sgnA!msg(red)
              fi
            fi
          :: to_sgnC!msg(yellow);
            if
            :: to_sgnA!msg(yellow);
              if
              :: to_sgnA!msg(red); to_sgnC!msg(red)
              :: to_sgnC!msg(red); to_sgnA!msg(red)
              fi
            :: to_sgnC!msg(red); to_sgnA!msg(yellow); to_sgnA!msg(red)
            fi
          fi;
                  ((stateA == red) && (stateC == red)) ->
                  to_sgnB!msg(blue); to_sgnD!msg(blue);

          ((stateA == red) && (stateB == blue)
          && (stateC == red) && (stateD == blue)) ->
          if
          :: to_sgnA!msg(yellow);
            if
            :: to_sgnB!msg(red); to_sgnD!msg(yellow); to_sgnD!msg(red)
            :: to_sgnD!msg(yellow);
              if
              :: to_sgnB!msg(red); to_sgnD!msg(red)
              :: to_sgnD!msg(red); to_sgnB!msg(red)
              fi
            fi
          :: to_sgnD!msg(yellow);
            if
            :: to_sgnB!msg(yellow);
              if
              :: to_sgnB!msg(red); to_sgnD!msg(red)
              :: to_sgnD!msg(red); to_sgnB!msg(red)
              fi
            :: to_sgnD!msg(red); to_sgnB!msg(yellow); to_sgnB!msg(red)
            fi
          fi;
          ((stateB == red) && (stateD == red)) ->
          to_sgnA!msg(blue); to_sgnC!msg(blue);
      od
}
```

検査する性質

```
    ![]p
p:((stateA == red||stateB == red)&&(stateB == red||stateC == red)
    &&(stateC == red||stateD == red)&&(stateD == red||stateA == red))
```

4.2 の 32 行目に誤りを混入させた。信号 B に出すはずのメッセージを信号 A に送っている。モデル検査の結果、次の反例が出力された。

リスト 4.3: 反例：時差式信号機

```
1  wl207086:2_9 ogawanaoya$ spin -t signal_time4.pml
2  Never claim moves to line 170   [(1)]
3            (blue,red,blue,red)
4                    (yellow,red,blue,red)
5                    (red,red,blue,red)
6                            (red,red,yellow,red)
7                            (red,red,red,red)
8                        (red,blue,red,red)
9                                (red,blue,red,blue)
10                   (yellow,blue,red,blue)
11 Never claim moves to line 169   [(!((((((stateA==red)||(stateB==red))
12                                 &&((stateB==red)||(stateC==red)))
13                                 &&((stateC==red)||(stateD==red)))
14                                 &&((stateD==red)||(stateA==red)))))]
15 Never claim moves to line 173   [(1)]
16 spin: trail ends after 71 steps
17 #processes: 6
18                 stateA = yellow
19                 stateB = blue
20                 stateC = red
21                 stateD = blue
22  71:    proc  5 (signalD) signal_time4.pml:138 (state 19)
23  71:    proc  4 (signalC) signal_time4.pml:122 (state 19)
24  71:    proc  3 (signalB) signal_time4.pml:108 (state 17)
25  71:    proc  2 (signalA) signal_time4.pml:90 (state 19)
26  71:    proc  1 (ctr) signal_time4.pml:57 (state 36)
27  71:    proc  0 (:init:) signal_time4.pml:164 (state 8) <valid end state>
28  71:    proc  - (never_0) signal_time4.pml:174 (state 8) <valid end state>
29 6 processes created
```

リスト 4.3 の 3 行目から 10 行目が変数 (A,B,C,D) の遷移である。3 行目から 4 行目の遷移から順に σ を使って変数を割り当てていく。9 行目から 10 行目の遷移の割り当ては次のようになる。$\sigma(A) = red, \sigma(A') = yellow, \sigma(B) = \sigma(B') = blue, \sigma(C) = \sigma(C') = red, \sigma(D) = \sigma(D') = blue$。$A' \neq red \wedge B' \neq red$ なので 9 行目から 10 行目の遷移が集合 u_1 の要素であるのでこの反例は負標本集合に含まれる。よって、モデルに誤りがあると判定できる。

4.3　実験

4.3.1　計画

負標本集合の評価観点として書きやすさと検出の十分性について評価する。評価の為の実験としてモデル検査の典型的な例であるリーダライタ問題、哲学者の食事問題、プリンタとスキャナの問題、時差式信号機の４つを取り上げた。

リーダライタ問題は書きやすさを評価する為の実験である。時差式信号機の例では故意に誤りを混入させたが、リーダライタ問題では故意でない誤りを含むモデルを用意し、書きやすさを評価する。

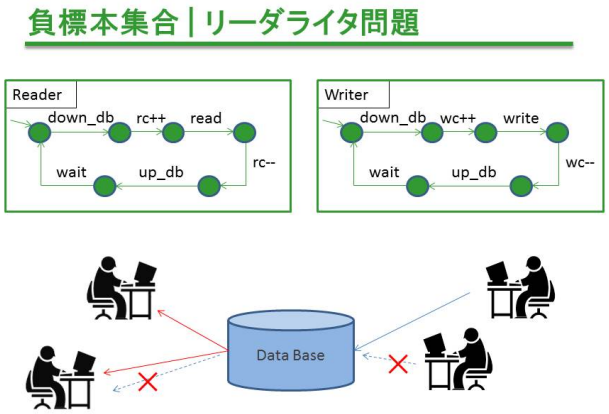
哲学者の食事問題は書きやすさを評価する為の実験である。モデル検査のよく知られている性質としてデッドロックと飢餓状態がある。デッドロックに関する振る舞いを負標本集合で書き負標本集合の記述の限界を知ることで評価する。

プリンタとスキャナの問題は書きやすさを評価するための実験である。

時差式信号機の例は検出の十分性を評価する為の実験である。信号機として起こりえない振る舞いについて負標本集合で書けるものと書けないものを示し検出の十分性を評価する。

4.3.2　実験内容

リーダライタ問題



データベースに複数の Reader と複数の Writer がアクセスする状況を考える。Reader は複数人が同時にデータベースにアクセスできるが、Writer がアクセスしているときは他のどの Writer も Reader もアクセスできない。振る舞いは、まずデータベースをロック状態にする。次に自分が Reader なら今アクセスしている Reader の数を、Writer なら今アクセスしている Writer の数を増やす。読み込みまたは書き込みが終わったら Reader、Writer の数を減らし、最後にデータベースをアンロック状態にする。よって起こりえない振る舞い

として『Writer がアクセス中のとき Reader の数が増える』、『Reader がアクセス中のとき Writer の数が増える』、『Reader または Writer がアクセス中のときデータベースがアンロック状態に変わる』、『データベースがロック状態に変わるとき既に Reader または Writer がアクセス中である』の 4 つを挙げる。この起こりえない振る舞いを集合 U を用いて記述する。変数の組は (db, rc, wc) とする。db はデータベースを表し、$locked, unlocked$ の値をとる。rc, wc はそれぞれ Reader と Writer の数を表し、整数値をとる。

まず『Writer がアクセス中のとき Reader の数が増える』は次のように書ける。

$$\boxed{u_1 \equiv (locked, rc, wc) \Rightarrow (locked, rc', wc') \ \ when \ \ wc \geq 1 \wedge rc' > rc}$$

$wc \geq 1$ で Writer がアクセス状態を、$rc' > rc$ で Reader が増えていることを表す。同様に『Reader がアクセス中のとき Writer の数が増える』は次のように書ける。

$$\boxed{u_2 \equiv (locked, rc, wc) \Rightarrow (locked, rc', wc') \ \ when \ \ rc \geq 1 \wedge wc' > wc}$$

『Reader または Writer がアクセス中のときデータベースがアンロック状態に変わる』は次のように書ける。

$$\boxed{u_3 \equiv (locked, rc, wc) \Rightarrow (unlocked, rc', wc') \ \ when \ \ rc \neq 0 \vee wc \neq 0}$$

『データベースがロック状態に変わるとき既に Reader または Writer がアクセス中である』は次のように書ける。

$$\boxed{u_4 \equiv (unlocked, rc, wc) \Rightarrow (locked, rc', wc') \ \ when \ \ rc \neq 0 \vee wc \neq 0}$$

Reader と Writer の振る舞いを表すモデルと検査する性質は次のようになる。但し検査する性質は表明で記述しリスト 4.5 の 5 行目に埋め込まれている。

リスト 4.4: Reader の振る舞いを表すモデル

```
1  active [2] proctype reader(){
2      do
3      :: if
4         :: d_step{(rc == 0) && (db == UNLOCKED)
5                   -> lock(db); write(db,rc,wc)}
6         ::!(rc == 0);
7         fi;
8      d_step{rc = rc + 1; write(db,rc,wc);}
9      /* read_data_base */
10     d_step{rc = rc - 1; write(db,rc,wc);}
11     :: if
12        :: d_step{(rc == 0) -> unlock(db); write(db,rc,wc)}
13        ::!(rc == 0);
14        fi
15     od
16 }
```

リスト 4.5: Writer の振る舞いを表すモデル

```
1  active [2] proctype writer(){
2      do
3      :: d_step{(rc == 0) && (db == UNLOCKED) -> lock(db); write(db,rc,wc)}
4         d_step{wc = wc + 1; write(db,rc,wc);}
5         label:
```

```
6        /* write_data_base */
7        d_step{wc = wc - 1; write(db,rc,wc);}
8        d_step{unlock(db); write(db,rc,wc)}
9      od
10 }
```

検査する性質

```
    [](p -> q)
p: writer@label
q: rc == 0
```

モデル検査の結果、次の反例が出力された。

リスト 4.6: 反例：リーダライタ問題

```
1  wl206082:RW ogawanaoya$ spin -t rw.pml
2                      (LOCKED,0,0)
3                      (LOCKED,0,1)
4              (UNLOCKED,0,1)
5              (LOCKED,0,1)
6              (LOCKED,1,1)
7  spin: rw.pml:47, Error: assertion violated
8  spin: text of failed assertion: assert((rc==0))
9  spin: trail ends after 6 steps
10 #processes: 4
11                  db = LOCKED
12                  rc = 1
13                  wc = 1
14   6:    proc  3 (writer) rw.pml:49 (state 17)
15   6:    proc  2 (writer) rw.pml:44 (state 23)
16   6:    proc  1 (reader) rw.pml:35 (state 20)
17   6:    proc  0 (reader) rw.pml:28 (state 31)
18 4 processes created
```

リスト 4.6 の 2 行目から 6 行目が変数 (db,rc,wc) の遷移である。3 行目から 4 行目の遷移が集合 U_3 の要素であるのでこの反例は負標本集合に含まれる。よって、モデルに誤りがあると判定できる。誤りの原因はリスト 4.4 の 11 行目のモデルのに意図しない非決定的な振る舞いが含まれていたことであった。なのでモデルを修正して再度、モデル検査を行なった。

リスト 4.7: 修正した Reader の振る舞いを表すモデル

```
1  active [2] proctype reader(){
2          do
3          ::if
4            ::d_step{(rc == 0) && (db == UNLOCKED)
5                    -> lock(db); write(db,rc,wc)}
6            ::!(rc == 0);
```

```
7            fi;
8            d_step{rc = rc + 1; write(db,rc,wc);}
9            /* read_data_base */
10           d_step{rc = rc - 1; write(db,rc,wc);}
11           if
12           :: d_step{(rc == 0) -> unlock(db); write(db,rc,wc)}
13           ::!(rc == 0);
14           fi
15         od
16 }
```

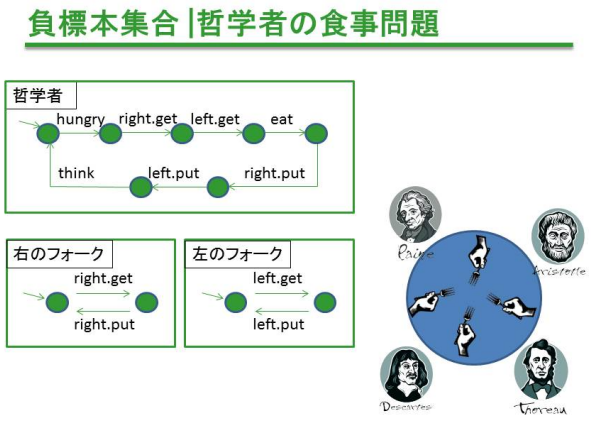
モデル検査の結果、次の反例が出力された。

リスト 4.8: 反例：リーダライタ問題

```
1  wl207219:RW ogawanaoya$ spin -t rw.pml
2              (LOCKED,0,0)
3              (LOCKED,1,0)
4              (LOCKED,0,0)
5              (UNLOCKED,0,0)
6                      (LOCKED,0,0)
7                      (LOCKED,0,1)
8        (LOCKED,1,1)
9  spin: rw.pml:47, Error: assertion violated
10 spin: text of failed assertion: assert((rc==0))
11 spin: trail ends after 9 steps
12 #processes: 4
13                 db = LOCKED
14                 rc = 1
15                 wc = 1
16   9:    proc  3 (writer) rw.pml:49 (state 17)
17   9:    proc  2 (writer) rw.pml:44 (state 23)
18   9:    proc  1 (reader) rw.pml:28 (state 29)
19   9:    proc  0 (reader) rw.pml:35 (state 19)
20 4 processes created
```

リスト 4.7 の 2 行目から 8 行目が変数 (db,rc,wc) の遷移である。7 行目から 8 行目の遷移が集合 U_1 の要素であるのでこの反例は負標本集合に含まれる。よって、モデルに誤りがあると判定できる。

哲学者の食事問題



4 人の哲学者が 4 本のフォークが置かれたテーブルを囲んでいる状況を考える。哲学者は自分の右側、左側の順にテーブルからフォークを取り、食事をする。食事が済んだら再びテーブルにフォークを戻す。よって起こりえない振る舞いとして『同じ哲学者がフォークを 3 つ所持していること』、『テーブルを介さずにフォークの所持者が変わること』、『全員がフォークを所持した状態である』、『ある哲学者は両方のフォークを所持することができない』の 4 つを挙げる。この起こりえない振る舞いを集合 U を用いて記述する。変数の組は (f_1, f_2, f_3, f_4) とする。f はフォークを表し、$T, 0, 1, 2, 3$ の値をとる。T はテーブルにあること、数字はどの哲学者が所持しているかを表す。

まず『同じ哲学者がフォークを 3 つ所持していること』は次のように書ける。

$$
\begin{aligned}
u_1 \equiv (f_0, f_1, f_2, f_3) \Rightarrow (f'_0, f'_1, f'_2, f'_3) \\
\quad when\ (f'_0 = f'_1 \wedge f'_1 = f'_2 \wedge f'_2 \neq T) \vee (f'_1 = f'_2 \wedge f'_2 = f'_3 \wedge f'_3 \neq T) \\
\vee (f'_2 = f'_3 \wedge f'_3 = f'_0 \wedge f'_0 \neq T) \vee (f'_3 = f'_0 \wedge f'_0 = f'_1 \wedge f'_1 \neq T)
\end{aligned}
$$

３本のフォークの値が同じだがテーブルには置かれていない状況を表している。

『テーブルを介さずにフォークの所持者が変わること』は次のように書ける。

$$
u_2 \equiv (f_0, f_1, f_2, f_3) \Rightarrow (f'_0, f'_1, f'_2, f'_3)\ when\ (f_n \neq f'_n \rightarrow f_n \neq T \wedge f'_n \neq T), n \in \{1, 2, 3, 4\}
$$

フォークの値が変化するのは誰かがテーブルから取った、またはテーブルに置いたときである。

『全員がフォークを所持した状態である』は次のように書ける。

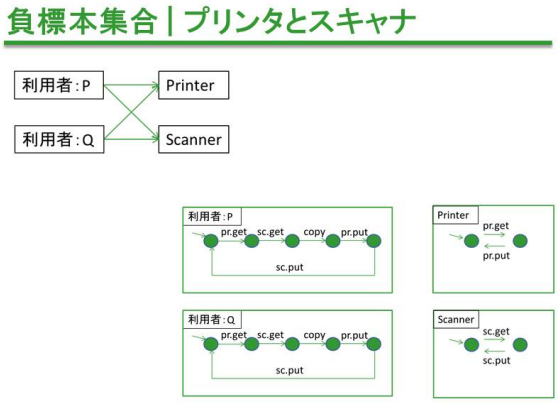
$$
\begin{aligned}
u_3 \equiv (f_0, f_1, f_2, f_3) \Rightarrow (f'_0, f'_1, f'_2, f'_3) \\
\quad when\ (f'_0 \neq T \wedge f'_1 \neq T \wedge f'_2 \neq T \wedge f'_3 \neq T) \\
\wedge (f'_0 \neq f'_1 \wedge f'_0 \neq f'_2 \wedge f'_0 \neq f'_3) \wedge (f'_1 \neq f'_2 \wedge f'_1 \neq f'_3) \wedge ((f'_2 \neq f'_3)
\end{aligned}
$$

どのフォークもテーブルにない状態で、かつどのフォークも同じ値を取らない状況を表している。

しかし『ある哲学者は両方のフォークを所持することができない』ことは負標本集合で記述できなかった。負標本集合は起こりえない振る舞いを変数の遷移で表すことで定義した

ので、変数の遷移が起こらない振る舞いに関しては記述できなかった。

プリンタとスキャナ



今、利用者 P と Q とが２つの資源 Printer と Scanner を使う状況を考える。利用者はスキャナから読み込んだデータをプリンタに出力する。この作業を行うためには Printer と Scanner を同時に必要とする。振る舞いとして『利用者の間で資源を手渡ししないこと』と、『利用者は先にプリンタを獲得すること』の２つを想定する。よって、起こりえない振る舞いとして『資源の解放が起こらすに利用者が代わること』と『スキャナを獲得したときその利用者がプリンタを獲得していないこと』の２つを挙げる。この起こりえない振る舞いを集合 U を用いて記述する。

まず『資源の解放が起こらすに利用者が代わること』は次のように書ける。

$$u_1 \equiv (pr, sc) \Rightarrow (pr', sc') \ \ when \ \ (pr \neq pr' \rightarrow pr \neq e \wedge pr' \neq e) \vee (sc \neq sc' \rightarrow sc \neq e \wedge sc' \neq e)$$

pr、pr'、sc、sc' は P, Q, e の値をとる変数である。pr の値が P であれば利用者 P がプリンタを獲得していることを表し、e であればプリンタが解放状態であることを表す。

『スキャナを獲得したときその利用者がプリンタを獲得していないこと』は次のように書ける。

$$u_2 \equiv (pr, sc) \Rightarrow (pr', sc') \ \ when \ \ (sc' \neq e \rightarrow pr' \neq sc')\}$$

利用者 P と Q の振る舞いを表すモデルと検査する性質は次のようになる。ただし、想定する振る舞いでは『利用者は先にプリンタを獲得すること』としたがリスト 4.9 では利用者 Q は先にスキャナを獲得するように 13 行目と 14 行目を入れ替えた。

リスト 4.9: プリンタとスキャナ

```
1  proctype userP(){
2          do
3          :: d_step{getP(pr); write(pr,sc)};
4             d_step{getP(sc); write(pr,sc)};
5             /* copy */
6             d_step{put(pr); write(pr,sc)};
```

```
7            d_step{put(sc); write(pr,sc)};
8          od
9  }
10
11 proctype userQ(){
12         do
13         ::d_step{getQ(sc); write(pr,sc)};
14           d_step{getQ(pr); write(pr,sc)};
15           /* copy */
16           d_step{put(pr); write(pr,sc)};
17           d_step{put(sc); write(pr,sc)};
18         od
19 }
```

検査する性質

```
    !(<>[]p)
p:(pr!=e && sc!=e)
```

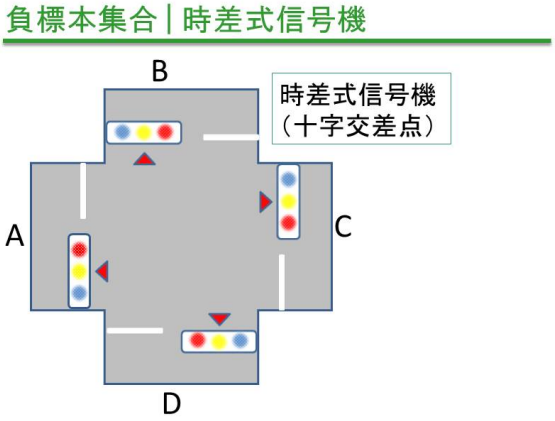
モデル検査の結果、次の反例が出力された。

リスト 4.10: 反例：プリンタとスキャナ

```
1  Never claim moves to line 58    [(1)]
2            (e,e)
3                    (e,Q)
4                    (Q,Q)
5                    (e,Q)
6                (P,Q)
7                    (P,e)
8                    (P,Q)
9  Never claim moves to line 57    [(((pr!=e)&&(sc!=e)))]
10 Never claim moves to line 62    [(((pr!=e)&&(sc!=e)))]
11 spin: trail ends after 21 steps
12 #processes: 3
13                  pr = P
14                  sc = Q
15  21:    proc  2 (userQ) prsc.pml:37 (state 14)
16  21:    proc  1 (userP) prsc.pml:27 (state 14)
17  21:    proc  0 (:init:) prsc.pml:51 (state 12) <valid end state>
18  21:    proc  - (never_0) prsc.pml:61 (state 9)
19 3 processes created
```

リスト 4.10 の 2 行目から 8 行目が変数 (pr,sc) の遷移である。7 行目から 8 行目の遷移が集合 u_2 の要素であるのでこの反例は負標本集合に含まれる。よって、モデルに誤りがあると判定できる。

時差式信号機



十字交差点での時差式信号機を考える。信号は左から時計回りに A,B,C,D とする。振る舞いは、まず信号 A と信号 C が青の状態から共に赤になる。このとき色が変化する順番は問わない。例えば信号 A が先に黄、赤と変化し次に信号 C が黄、赤と変化することもあれば信号 A が黄、信号 C が黄、赤、信号 A が赤と変化することもある。共に赤になったら次は信号 B と D が青になる。よって起こりえない振る舞いは『隣り合う信号がどちらも赤ではない』、『各信号が青⇒黄⇒赤の順に変化しない』、『ある信号が赤になったとき向い側の信号が赤になる前に変化する』の 3 つを挙げる。この起こりえない振る舞いを集合 U を用いて記述する。変数の組は (A, B, C, D) とする。A, B, C, D は各信号を表し、$blue, yellow, red$ の値をとる。

まず『隣り合う信号がどちらも赤ではない』は次のように書ける。

$$u_1 \equiv (A, B, C, D) \Rightarrow (A', B', C', D') \quad when \ (A' \neq red \wedge B' \neq red) \vee (B' \neq red \wedge C' \neq red) \vee (C' \neq red \wedge D' \neq red) \vee (D' \neq red \wedge A' \neq red)$$

『各信号が青⇒黄⇒赤の順に変化しない』は次のように書ける。

$$u_2 \equiv (A, B, C, D) \Rightarrow (A', B', C', D') \quad when \ (A = blue \wedge A' = red) \vee (A = yellow \wedge A' = blue) \vee (A = red \wedge A' = yellow)$$

$$u_3 \equiv (A, B, C, D) \Rightarrow (A', B', C', D') \quad when \ (B = blue \wedge B' = red) \vee (B = yellow \wedge B' = blue) \vee (B = red \wedge B' = yellow)$$

$$u_4 \equiv (A, B, C, D) \Rightarrow (A', B', C', D') \quad when \ (C = blue \wedge C' = red) \vee (C = yellow \wedge C' = blue) \vee (C = red \wedge C' = yellow)$$

$$u_5 \equiv (A, B, C, D) \Rightarrow (A', B', C', D') \quad when \ (D = blue \wedge D' = red) \vee (D = yellow \wedge D' = blue) \vee (D = red \wedge D' = yellow)$$

『ある信号が赤になったとき向い側の信号が赤になる前に変化する』は負標本集合では記述できなかった。なので、本研究で定義した負標本集合の書き方では記述できないことを示すために次の実験を行なった。

リスト 4.11: 時差式信号機の振る舞いを表すモデル

```
proctype ctr(chan to_sgnA, to_sgnB, to_sgnC, to_sgnD)
{
        again1:
        if
        :: if
          :: to_sgnB!msg(yellow);
                if
                :: to_sgnB!msg(red);
                  again2:
                  if
                  :: to_sgnB!msg(blue); goto again1;
                  :: to_sgnD!msg(yellow); to_sgnD!msg(red);
                        if
                        :: goto again2;
                        :: goto again3;
                        fi;
                  fi
                :: to_sgnD!msg(yellow);
                  if
                  :: to_sgnB!msg(red);
                  :: to_sgnD!msg(red);
                  fi;
                  if
                  :: goto again2;
                  :: goto again3;
                  fi;
                fi
          :: to_sgnD!msg(yellow);
                if
                :: to_sgnD!msg(red);
                  again3:
                  if
                  :: to_sgnD!msg(blue); goto again1;
                  :: to_sgnB!msg(yellow); to_sgnB!msg(red);
                        if
                        :: goto again2;
                        :: goto again3;
                        fi;
                  fi
                :: to_sgnB!msg(yellow);
                  if
                  :: to_sgnB!msg(red);
                  :: to_sgnD!msg(red);
                  fi;
                  if
                  :: goto again2;
                  :: goto again3;
                  fi;
                fi;
          fi;
        fi
```

```
}
```

検査する性質

```
[](r → (p U q))          p:(B == red)
                         q:(D == red)
                         r:(B == red)
```

リスト 4.11 は『隣り合う信号がどちらも赤ではない』、『各信号が青⇒黄⇒赤の順に変化しない』振る舞いは起こらないが『ある信号が赤になったとき向い側の信号が赤になる前に変化する』振る舞いは起こるモデルである。性質は信号 B が赤になったとき信号 D が赤になるまでは信号 B は赤のままであることを検査した。モデル検査の結果、次の反例が出力された。

リスト 4.12: 反例：時差式信号機

```
1  wl207051:2_6 ogawanaoya$ spin -t signal_again.pml
2  Never claim moves to line 158   [(1)]
3            (red,blue,red,blue)
4                        (red,yellow,red,blue)
5                        (red,red,red,blue)
6  Never claim moves to line 156   [((!((stateD==red))&&(stateB==red)))]
7  Never claim moves to line 162   [(!((stateD==red)))]
8                        (red,blue,red,blue)
9                        (red,yellow,red,blue)
10                       (red,red,red,blue)
11 spin: trail ends after 50 steps
12 #processes: 6
13                 stateA = red
14                 stateB = red
15                 stateC = red
16                 stateD = blue
17  50:    proc  5 (signalD) signal_again.pml:122 (state 19)
18  50:    proc  4 (signalC) signal_again.pml:106 (state 19)
19  50:    proc  3 (signalB) signal_again.pml:92 (state 17)
20  50:    proc  2 (signalA) signal_again.pml:74 (state 19)
21  50:    proc  1 (ctr) signal_again.pml:20 (state 49)
22  50:    proc  0 (:init:) signal_again.pml:150 (state 13) <valid end state>
23  50:    proc  - (never_0) signal_again.pml:161 (state 13)
24 6 processes created
```

リスト 4.12 の 3 行目から 5 行目が変数 (A,B,C,D,) の遷移である。『ある信号が赤になったとき向い側の信号が赤になる前に変化する』振る舞いを変数の遷移で表現するには 3 行目から 4 行目の遷移と 4 行目から 5 行目の遷移の 2 遷移が必要となる。

4.3.3 評価

典型的な例としてリーダライタ問題、哲学者の食事問題、時差式信号機を取り上げた。評価は、各々の実験から書きやすさと検出の十分性について評価する。

リーダライタ問題の例では故意でない誤りを含むモデルを用意した。リスト 4.4 では意図しない非決定的な振る舞いが含まれていたことが誤りの原因であるが、詳しく説明する。:: は非決定的な振る舞いを記述するときに用いる。3 行目から 7 行目で、まずリーダはデータベースを操作するとき、rc が 0 ならデータベースをロックする。rc が 0 でない、すなわち他のリーダが操作中のときには既にライタが操作できないようにロックされているので次の処理に進んでいく。11 行目から 14 行目で、最後にデータベースのロックを解除するわけだが rc が 0、すなわち自分の他にリーダがいないときには解除し、そうでなければ解除はせず再び 3 行目からの処理に進む。データベースをロック、解除するときは rc の値によって非決定的な実行が必要であるが、データベースのロック、解除自体は非決定的に実行されることではない。しかし 11 行目で :: if とあるように非決定的な処理をしてしまっている。8 行目から 10 行目でインデントを変えてしまったことで、11 行目に書く処理は 3 行目と非決定的に実行されるものだと勘違いしてしまったことが誤りの原因と考えられる。このように些細なことが原因でモデルに誤りが混入する場合がある。一方、負標本集合は着目した変数の条件を考えることで書くことができる。リーダライタ問題では、起こりえない振る舞いの 1 つとして『Reader または Writer がアクセス中のときデータベースがアンロック状態に変わる』ことを挙げた。なので、リーダの数を表す rc、ライタの数を表す wc、データベースの状態を表す db に着目すれば良い。リーダ、ライタの振る舞いではデータベースをロックしてから rc, wc の数を増やす。よって、$u_3 \equiv (locked, rc, wc) \Rightarrow (unlocked, rc', wc')\ \ when\ \ rc \neq 0 \vee wc \neq 0$、と書けた。モデルを作成していく過程よりも負標本集合を書く過程のほうが、時間的にも量的にも短く書きやすかった。リスト 4.7 はリスト 4.4 で述べた誤りだけを修正したモデルであった。リスト 4.7 では不可分実行として指定する範囲が狭かったことが誤りの原因であるが、詳しく説明する。今、２人のリーダがデータベースを操作する状況を考える。１人目のリーダは、4 行目でデータベースをロックした後、8 行目で rc を増やしたところまで処理が進んだとする。２人目のリーダは、既に他のリーダがいるので 8 行目の手前まで処理が進んだとする。２人目のリーダはまだ rc を増やしていないので、この状況で 1 人目のリーダが rc を減らし、次の処理に進むと rc は 0 となり、自分が最後のリーダだと判断し、データベースをライタが操作可能な状態にしてしまう。この問題を解決するには 8 行目の処理までを 4 行目及び、6 行目の処理とまとめる必要があると考えられる。こちらも、負標本集合は着目した変数の条件を考えることで書くことができる。リーダライタ問題では、起こりえない振る舞いの 1 つとして『Writer がアクセス中のとき Reader の数が増える』ことを挙げた。なので、リーダの数を表す rc、ライタの数を表す wc に着目すれば良い。よって、$u_1 \equiv (locked, rc, wc) \Rightarrow (locked, rc', wc')\ \ when\ \ wc \geq 1 \wedge rc' > rc$、と書けた。モデルの意味に注意したり、リーダの並行動作を考えてモデルを作成するよりも、着目した変数の条件を考える方が書きやすかった。モデルは振る舞い全体を把握して書くのに比べて、負標本

集合は振る舞いの一部を書けば良かったので書きやすかった。

哲学者の食事問題では、起こりえない振る舞いとして『同じ哲学者がフォークを 3 つ所持していること』、『テーブルを介さずにフォークの所持者が変わること』、『全員がフォークを所持した状態である』、『ある哲学者は両方のフォークを所持することができない』、『ある哲学者は両方のフォークを所持することができない』ことを挙げた。『同じ哲学者がフォークを 3 つ所持していること』と『テーブルを介さずにフォークの所持者が変わること』は明らかに起こりえない振る舞いだと考えられる。3 本のフォークに着目し、全て同じ値をとることで表現できる。あとは変数の添字を変えればよい。よって、$u_1 \equiv (f_0, f_1, f_2, f_3) \Rightarrow (f'_0, f'_1, f'_2, f'_3)\ when\ (f'_0 = f'_1 \wedge f'_1 = f'_2 \wedge f'_2 \neq T) \vee (f'_1 = f'_2 \wedge f'_2 = f'_3 \wedge f'_3 \neq T) \vee (f'_2 = f'_3 \wedge f'_3 = f'_0 \wedge f'_0 \neq T) \vee (f'_3 = f'_0 \wedge f'_0 = f'_1 \wedge f'_1 \neq T)$、と書けた。『テーブルを介さずにフォークの所持者が変わること』はフォークの値が変わったときは、テーブルから取ったか、テーブルに置いたかのどちらかなので、$u_2 \equiv (f_0, f_1, f_2, f_3) \Rightarrow (f'_0, f'_1, f'_2, f'_3)\ when\ (f_n \neq f'_n \rightarrow f_n \neq T \wedge f'_n \neq T), n \in \{1, 2, 3, 4\}$、と書けた。いずれも自然言語から手順を踏むことで書くことができた。『ある哲学者は両方のフォークを所持することができない』、『ある哲学者は両方のフォークを所持することができない』ことについては考察で述べる。

プリンタとスキャナの例で起こりえない振る舞いの 1 つとして『資源の解放が起こらすに利用者が代わること』を挙げ記述することができた。これは哲学者の食事問題の例だと『テーブルを介さずにフォークの所持者が変わること』に相当する。このように異なる例でも一度、負標本集合が書けていればそれを参考にして別の例でも用いることができる点が良かった。

時差式信号機の例で、『隣り合う信号がどちらも赤ではない』ことと、『各信号が青⇒黄⇒赤の順に変化しない』ことは書くことができたが、『ある信号が赤になったとき向い側の信号が赤になる前に変化する』ことは書けなかった。『隣り合う信号がどちらも赤ではない』ことは信号の遷移とは関係なく書ける。遷移した先で隣り合う信号のどちらも赤でないときが起こりえない振る舞いであり、遷移する前は関係ない。『各信号が青⇒黄⇒赤の順に変化しない』ことを書くには遷移が関係する。信号は青⇒黄⇒赤の順に変化するので遷移前が青なら遷移後が赤なら起こりえない振る舞いとなる。『ある信号が赤になったとき向い側の信号が赤になる前に変化する』ことも遷移が関係する。しかし、ある信号について赤になったことを書くために 1 遷移、さらに向い側の信号の赤を待たずに変化することを書くために 1 遷移必要である。変数の遷移で表現するには 2 遷移必要だった。なので負標本集合では記述できなかった。本研究の書き方だと、1 遷移までなら検出できる。

4.3.4 考察

共通の利点は負標本集合を用いることで『モデル化』に誤りがあることが明らかになることである。リーダライタ問題の例を取り上げる。性質 P は表明で記述したが、かなり単純な形で記述できた。反例が検出されたとき性質 P は単純であるため、モデル化または

真のモデルに誤りがあると見当がつく。反例が負標本集合に含まれていれば、モデル M は起こりえない振る舞いを含んでいることになるので「モデル化」に誤りがあることが明らかになった。

リーダライタ問題で、Reader や Writer がどう並行に動作するかはモデルを書く過程では把握しにくい。一方で負標本集合で書くことは変数の値だけに着目すれば良いので整理しやすい。よって負標本集合はミスなく書けたと考えられる。

哲学者の食事問題で、『同じ哲学者がフォークを 3 つ所持していること』と『テーブルを介さずにフォークの所持者が変わること』は明らかに起こりえない振る舞いだと考えられる。しかし、『全員がフォークを所持した状態である』ことは負標本集合で書くべきことではない。哲学者の食事問題とはデッドロックに陥る例として、たびたび用いられる。むしろ『いつまでも全員がフォークを所持した状態にならない』ことを起こりえない振る舞いとして挙げるべきだったかもしれない。しかし『ある哲学者は両方のフォークを所持することができない』ことが負標本集合では書きようがなかったことから『いつまでも全員がフォークを所持した状態にならない』ことも負標本集合では書けないと考えられる。理由はいずれも飢餓状態を表現しようとしており、デッドロックは起こってほしくない、ある状態への遷移で表現できるのに対し、飢餓状態は一向にある状態にならないことを表現しなくてはならない。よって飢餓状態は負標本集合では記述できなかったと考えられる。また、デッドロックは記述できたが饑餓状態は記述できなかったことから安全性[2] は負標本集合で記述できるが活性[3] は記述できないと考えられる。

プリンタとスキャナの例で、起こりえない振る舞いの 1 つとして『資源の解放が起こらすに利用者が代わること』を挙げ記述することができた。これは哲学者の食事問題の負標本集合を参考にして書くことができる。このように異なる例でも一度、負標本集合が書けていればそれを参考にして別の例でも用いることができたが、検査対象のモデル化を行なうときでも同様のことがいえるかもしれない。しかし、モデルのどこが対応しているか比較していくよりも条件式を比較したほうがわかりやすいだろう。よって、他の検査対象を参考できるなら、モデル化よりも負標本集合を書くほうが書きやすいと考えられる。今回、利用者 P と Q の２人だけであったり、混入させたモデルの誤りは単純なものであった。しかし利用者が増えたり、複雑な処理を想定したモデルを作成する場合でも、資源が変わらなければ同じ負標本集合を用いることができると考えられる。

時差式信号の例で、『ある信号が赤になったとき向い側の信号が赤になる前に変化する』振る舞いは最初の「ある信号」の変化についてどうしても 2 遷移必要なので今回定義した負標本集合の書き方では書けなかった。また、この振る舞いは向い側の信号が飢餓状態に陥っていることの言い換えである。よって哲学者の食事問題の考察より、やはり負標本集合では記述できないと考えられる。

今回、書きやすさとして振る舞いの一部分を書けば良いこと、変数にだけ着目することについて評価した。その他に使用する変数の個数を評価する方法が考えられる。モデルで

[2]悪い事は起きないことを保証する性質。
[3]いつか良いことが起きることを保証する性質。

使用する変数と負標本集合で使用する変数を比較できれば定量的な評価が得られると考えられる。

第5章　正標本集合による誤り特定の実験

5.1　実験

5.1.1　計画

正標本集合による誤り特定の実験は正標本集合の書き方について考察するために行なう。正標本集合に関しては現時点で明確な書き方は見つからなかった。実験は２つ行なう。まずは故意に性質 P に誤りを混入させて、その性質 P の誤りを特定する実験である。仮に正標本集合は変数の組の遷移列 U' で表すことにする。負標本集合とは違い遷移は 1 遷移とは限らない。実験は時差式信号機の例を用いる。２つめはプリンタとスキャナの問題を用いる。ひとつの誤ったモデルに対して正しい性質と誤った性質を与えて検出される反例の違いから正標本集合の書き方を探る。

5.1.2　実験内容

時差式信号機

4.3.2 の時差式信号機の例と同様の状況を考える。性質は『隣り合う信号のどちらかが赤である』、『各信号が青⇒黄⇒赤の順に変化する』、『ある信号が赤になったとき隣り合う信号が赤になるまではずっと赤である』の３つを検査する。ここで正標本集合として信号 A,B,C,D の色の４つ組で書いておく。正標本集合は (B, R, B, R) から (R, B, R, B) に変化する６通りと (B, R, B, R) から (R, B, R, B) に変化する６通りの計１２通りである。B は *blue*、Y は *yellow*、R は *red* を意味する。

$$
\begin{aligned}
U'_1 &= (B,R,B,R) \Rightarrow (Y,R,B,R) \Rightarrow (R,R,B,R) \Rightarrow (R,R,Y,R) \Rightarrow (R,R,R,R) \Rightarrow (R,B,R,B) \\
U'_2 &= (B,R,B,R) \Rightarrow (Y,R,B,R) \Rightarrow (Y,R,Y,R) \Rightarrow (R,R,Y,R) \Rightarrow (R,R,R,R) \Rightarrow (R,B,R,B) \\
U'_3 &= (B,R,B,R) \Rightarrow (Y,R,B,R) \Rightarrow (Y,R,Y,R) \Rightarrow (Y,R,R,R) \Rightarrow (R,R,R,R) \Rightarrow (R,B,R,B) \\
U'_4 &= (B,R,B,R) \Rightarrow (B,R,Y,R) \Rightarrow (B,R,R,R) \Rightarrow (Y,R,R,R) \Rightarrow (R,R,R,R) \Rightarrow (R,B,R,B) \\
U'_5 &= (B,R,B,R) \Rightarrow (B,R,Y,R) \Rightarrow (Y,R,Y,R) \Rightarrow (Y,R,R,R) \Rightarrow (R,R,R,R) \Rightarrow (R,B,R,B) \\
U'_6 &= (B,R,B,R) \Rightarrow (B,R,Y,R) \Rightarrow (Y,R,Y,R) \Rightarrow (R,R,Y,R) \Rightarrow (R,R,R,R) \Rightarrow (R,B,R,B) \\
U'_7 &= (R,B,R,B) \Rightarrow (R,Y,R,B) \Rightarrow (R,R,R,B) \Rightarrow (R,R,R,Y) \Rightarrow (R,R,R,R) \Rightarrow (B,R,B,R) \\
U'_8 &= (R,B,R,B) \Rightarrow (R,Y,R,B) \Rightarrow (R,Y,R,Y) \Rightarrow (R,R,R,Y) \Rightarrow (R,R,R,R) \Rightarrow (B,R,B,R) \\
U'_9 &= (R,B,R,B) \Rightarrow (R,Y,R,B) \Rightarrow (R,Y,R,Y) \Rightarrow (R,Y,R,R) \Rightarrow (R,R,R,R) \Rightarrow (B,R,B,R) \\
U'_{10} &= (R,B,R,B) \Rightarrow (R,B,R,Y) \Rightarrow (R,B,R,R) \Rightarrow (R,Y,R,R) \Rightarrow (R,R,R,R) \Rightarrow (B,R,B,R) \\
U'_{11} &= (R,B,R,B) \Rightarrow (R,B,R,Y) \Rightarrow (R,Y,R,Y) \Rightarrow (R,Y,R,R) \Rightarrow (R,R,R,R) \Rightarrow (B,R,B,R) \\
U'_{12} &= (R,B,R,B) \Rightarrow (R,B,R,Y) \Rightarrow (R,Y,R,Y) \Rightarrow (R,R,R,Y) \Rightarrow (R,R,R,R) \Rightarrow (B,R,B,R)
\end{aligned}
$$

信号機はコントローラを介したメッセーッジ通信で表現した。信号機とコントローラの振る舞いを表すモデルと検査する性質は次のようになる。

リスト 5.1: 信号機の振る舞いを表すモデル

```
proctype signalA(chan to_ctr)
{
    mtype state;
    do
    :: to_ctr?msg(state);
        if
            ::(state == blue) ->
              d_step{stateA = blue; write(stateA,stateB,stateC,stateD)}
            ::(state == yellow) ->
              d_step{stateA = yellow; write(stateA,stateB,stateC,stateD)}
            ::(state == red) ->
              d_step{stateA = red; write(stateA,stateB,stateC,stateD)}
        fi
    od
}
```

リスト 5.2: コントローラの振る舞いを表すモデル

```
proctype ctr(chan to_sgnA, to_sgnB, to_sgnC, to_sgnD)
{
    do
    ::((stateA == blue) && (stateB == red)
       && (stateC == blue) && (stateD == red)) ->
        if
        ::to_sgnA!msg(yellow);
          if
          ::to_sgnA!msg(red); to_sgnC!msg(yellow); to_sgnC!msg(red)
          ::to_sgnC!msg(yellow);
            if
            ::to_sgnA!msg(red); to_sgnC!msg(red)
```

```
13                ::to_sgnC!msg(red); to_sgnA!msg(red)
14                fi
15              fi
16            ::to_sgnC!msg(yellow);
17              if
18              ::to_sgnA!msg(yellow);
19                if
20                ::to_sgnA!msg(red); to_sgnC!msg(red)
21                ::to_sgnC!msg(red); to_sgnA!msg(red)
22                fi
23              ::to_sgnC!msg(red); to_sgnA!msg(yellow); to_sgnA!msg(red)
24              fi
25            fi;
26                    ((stateA == red) && (stateC == red)) ->
27                    to_sgnB!msg(blue); to_sgnD!msg(blue);
28
29            ((stateA == red) && (stateB == blue)
30            && (stateC == red) && (stateD == blue)) ->
31            if
32            ::to_sgnB!msg(yellow);
33              if
34              ::to_sgnB!msg(red); to_sgnD!msg(yellow); to_sgnD!msg(red)
35              ::to_sgnD!msg(yellow);
36                if
37                ::to_sgnB!msg(red); to_sgnD!msg(red)
38                ::to_sgnD!msg(red); to_sgnB!msg(red)
39                fi
40              fi
41            ::to_sgnD!msg(yellow);
42              if
43              ::to_sgnB!msg(yellow);
44                if
45                ::to_sgnB!msg(red); to_sgnD!msg(red)
46                ::to_sgnD!msg(red); to_sgnB!msg(red)
47                fi
48              ::to_sgnD!msg(red); to_sgnB!msg(yellow); to_sgnB!msg(red)
49              fi
50            fi;
51            ((stateB == red) && (stateD == red)) ->
52            to_sgnA!msg(blue); to_sgnC!msg(blue);
53        od
54    }
```

『隣り合う信号のどちらかが赤である』ことを検査する為に次の LTL 式を与えた。隣り合う信号のパターンは４通りある。

検査する性質

```
[]p
  p:((stateA == red)||(stateB == red))
  p:((stateB == red)||(stateC == red))
  p:((stateC == red)||(stateD == red))
  p:((stateD == red)||(stateA == red))
```

モデル検査の結果、いずれも反例は出力されなかった。

『各信号が青⇒黄⇒赤の順に変化する』ことを検査する為に次のLTL式を与えた。「信号Aが青になったとき、黄になるまではずっと青」と表現した。色の変化で３通り、各信号４通りなのでr,p,qの組み合わせは１２通りある。

検査する性質

```
r → [](p U q)
  r:(stateA == blue),  p:(stateA == blue),  q:(stateA == yellow)
  r:(stateB == blue),  p:(stateB == blue),  q:(stateB == yellow)
  r:(stateC == blue),  p:(stateC == blue),  q:(stateC == yellow)
  r:(stateD == blue),  p:(stateD == blue),  q:(stateD == yellow)
  r:(stateA == yellow),p:(stateA == yellow),q:(stateA == red)
  r:(stateB == yellow),p:(stateB == yellow),q:(stateB == red)
  r:(stateC == yellow),p:(stateC == yellow),q:(stateC == red)
  r:(stateD == yellow),p:(stateD == yellow),q:(stateD == red)
  r:(stateA == red),   p:(stateA == red),   q:(stateA == blue)
  r:(stateB == red),   p:(stateB == red),   q:(stateB == blue)
  r:(stateC == red),   p:(stateC == red),   q:(stateC == blue)
  r:(stateD == red),   p:(stateD == red),   q:(stateD == blue)
```

モデル検査の結果、次の４つの反例が出力された。

リスト 5.3: 反例：信号機A(青⇒黄)

```
wl205003:9_03 ogawanaoya$ spin -t signal_time4.pml
Never claim moves to line 173   [((stateA==blue))]
          (blue,red,blue,red)
Never claim moves to line 184   [(1)]
                    (yellow,red,blue,red)
                    (red,red,blue,red)
Never claim moves to line 182   [(!((stateA==yellow)))]
Never claim moves to line 177   [(!((stateA==yellow)))]
                            (red,red,yellow,red)
                            (red,red,red,red)
                        (red,blue,red,red)
```

```
                                (red,blue,red,blue)
                        (red,yellow,red,blue)
                        (red,red,red,blue)
                                (red,red,red,yellow)
                                (red,red,red,red)
                            (red,red,blue,red)
Never claim moves to line 178   [((!((stateA==blue))&&!((stateA==yellow))))]
                    (blue,red,blue,red)
Never claim moves to line 187   [(1)]
spin: trail ends after 105 steps
#processes: 6
                stateA = blue
                stateB = red
                stateC = blue
                stateD = red
105:    proc  5 (signalD) signal_time4.pml:139 (state 19)
105:    proc  4 (signalC) signal_time4.pml:123 (state 19)
105:    proc  3 (signalB) signal_time4.pml:107 (state 19)
105:    proc  2 (signalA) signal_time4.pml:91 (state 19)
105:    proc  1 (ctr) signal_time4.pml:19 (state 65)
105:    proc  0 (:init:) signal_time4.pml:165 (state 8) <valid end state>
105:    proc  - (never_0) signal_time4.pml:188 (state 24) <valid end state>
6 processes created
```

リスト 5.4: 反例：信号機 B(赤⇒青)

```
wl205003:9_03 ogawanaoya$ spin -t signal_time4.pml
Never claim moves to line 173   [((stateB==red))]
          (blue,red,blue,red)
Never claim moves to line 184   [(1)]
                    (yellow,red,blue,red)
                    (red,red,blue,red)
                            (red,red,yellow,red)
                            (red,red,red,red)
                        (red,blue,red,red)
                                (red,blue,red,blue)
                        (red,yellow,red,blue)
Never claim moves to line 182   [(!((stateB==blue)))]
Never claim moves to line 177   [(!((stateB==blue)))]
                                (red,yellow,red,yellow)
                                (red,yellow,red,red)
Never claim moves to line 178   [((!((stateB==red))&&!((stateB==blue))))]
                        (red,red,red,red)
Never claim moves to line 187   [(1)]
spin: trail ends after 90 steps
#processes: 6
                stateA = red
                stateB = red
                stateC = red
                stateD = red
 90:    proc  5 (signalD) signal_time4.pml:139 (state 19)
 90:    proc  4 (signalC) signal_time4.pml:123 (state 19)
```

```
 90:     proc  3 (signalB) signal_time4.pml:107 (state 19)
 90:     proc  2 (signalA) signal_time4.pml:91 (state 19)
 90:     proc  1 (ctr) signal_time4.pml:82 (state 62)
 90:     proc  0 (:init:) signal_time4.pml:165 (state 8) <valid end state>
 90:     proc  - (never_0) signal_time4.pml:188 (state 24) <valid end state>
6 processes created
```

リスト 5.5: 反例：信号機 C(青⇒黄)

```
wl205003:9_03 ogawanaoya$ spin -t signal_time4.pml
Never claim moves to line 173   [((stateC==blue))]
          (blue,red,blue,red)
Never claim moves to line 184   [(1)]
                   (yellow,red,blue,red)
                   (red,red,blue,red)
                           (red,red,yellow,red)
                           (red,red,red,red)
Never claim moves to line 182   [(!((stateC==yellow)))]
Never claim moves to line 177   [(!((stateC==yellow)))]
                       (red,blue,red,red)
                               (red,blue,red,blue)
                       (red,yellow,red,blue)
                       (red,red,red,blue)
                               (red,red,red,yellow)
                               (red,red,red,red)
                   (blue,red,red,red)
Never claim moves to line 178   [((!((stateC==blue))&&!((stateC==yellow))))]
                           (blue,red,blue,red)
Never claim moves to line 187   [(1)]
spin: trail ends after 105 steps
#processes: 6
                 stateA = blue
                 stateB = red
                 stateC = blue
                 stateD = red
105:    proc  5 (signalD) signal_time4.pml:139 (state 19)
105:    proc  4 (signalC) signal_time4.pml:123 (state 19)
105:    proc  3 (signalB) signal_time4.pml:107 (state 19)
105:    proc  2 (signalA) signal_time4.pml:91 (state 19)
105:    proc  1 (ctr) signal_time4.pml:19 (state 65)
105:    proc  0 (:init:) signal_time4.pml:165 (state 8) <valid end state>
105:    proc  - (never_0) signal_time4.pml:188 (state 24) <valid end state>
6 processes created
```

リスト 5.6: 反例：信号機 D(赤⇒青)

```
wl205003:9_03 ogawanaoya$ spin -t signal_time4.pml
Never claim moves to line 173   [((stateD==red))]
          (blue,red,blue,red)
Never claim moves to line 184   [(1)]
                   (yellow,red,blue,red)
                   (red,red,blue,red)
```

```
                                (red,red,yellow,red)
                                (red,red,red,red)
                            (red,blue,red,red)
                                    (red,blue,red,blue)
                            (red,yellow,red,blue)
                            (red,red,red,blue)
                                    (red,red,red,yellow)
Never claim moves to line 182   [(!((stateD==blue)))]
Never claim moves to line 177   [(!((stateD==blue)))]
Never claim moves to line 178   [((!((stateD==red))&&!((stateD==blue))))]
                                    (red,red,red,red)
Never claim moves to line 187   [(1)]
spin: trail ends after 90 steps
#processes: 6
                stateA = red
                stateB = red
                stateC = red
                stateD = red
 90:    proc  5 (signalD) signal_time4.pml:139 (state 19)
 90:    proc  4 (signalC) signal_time4.pml:123 (state 19)
 90:    proc  3 (signalB) signal_time4.pml:107 (state 19)
 90:    proc  2 (signalA) signal_time4.pml:91 (state 19)
 90:    proc  1 (ctr) signal_time4.pml:82 (state 62)
 90:    proc  0 (:init:) signal_time4.pml:165 (state 8) <valid end state>
 90:    proc  - (never_0) signal_time4.pml:188 (state 24) <valid end state>
6 processes created
```

これらの反例は正標本集合同士の組み合わせ、ないしは一部分を組み合わせたものである。いずれも本来、正しいはずの変数の遷移が反例として検出されている。よって性質の与え方に誤りが含まれると特定できる。

次に以下の LTL 式で同様に『各信号が青⇒黄⇒赤の順に変化する』ことを検査した。

検査する性質

```
[](r → (p U q))
  r:(stateA == blue),  p:(stateA == blue),  q:(stateA == yellow)
  r:(stateB == blue),  p:(stateB == blue),  q:(stateB == yellow)
  r:(stateC == blue),  p:(stateC == blue),  q:(stateC == yellow)
  r:(stateD == blue),  p:(stateD == blue),  q:(stateD == yellow)
  r:(stateA == yellow),p:(stateA == yellow),q:(stateA == red)
  r:(stateB == yellow),p:(stateB == yellow),q:(stateB == red)
  r:(stateC == yellow),p:(stateC == yellow),q:(stateC == red)
  r:(stateD == yellow),p:(stateD == yellow),q:(stateD == red)
  r:(stateA == red),   p:(stateA == red),   q:(stateA == blue)
  r:(stateB == red),   p:(stateB == red),   q:(stateB == blue)
  r:(stateC == red),   p:(stateC == red),   q:(stateC == blue)
  r:(stateD == red),   p:(stateD == red),   q:(stateD == blue)
```

モデル検査の結果、いずれも反例は出力されなかった。
『ある信号が赤になったとき隣り合う信号が赤になるまではずっと赤である』ことを検査する為に次の LTL 式を与えた。

検査する性質

```
[](r → (p U q))
  r:(stateA == red), p:(stateA == red), q:(stateB == red)
  r:(stateA == red), p:(stateA == red), q:(stateD == red)
  r:(stateB == red), p:(stateB == red), q:(stateA == red)
  r:(stateB == red), p:(stateB == red), q:(stateC == red)
  r:(stateC == red), p:(stateC == red), q:(stateB == red)
  r:(stateC == red), p:(stateC == red), q:(stateD == red)
  r:(stateD == red), p:(stateD == red), q:(stateA == red)
  r:(stateD == red), p:(stateD == red), q:(stateC == red)
```

モデル検査の結果、いずれも反例は出力されなかった。

プリンタとスキャナ

プリンタとスキャナの例ではリスト 4.9 のモデルを用いる。このデッドロックを起こすモデルに対して誤った LTL 式 P_1 と正しい LTL 式 P_2 を与える。LTL 式は共にデッドロックを起こさない意味合いで記述した。

誤った LTL 式

```
[](p → q)
  p:(pr != e && sc != e)
  q:(pr == sc)
```

正しい LTL 式

```
!(<>[]p)
  p:(pr != e && sc != e)
```

P_1 は資源が共に獲得中になったとき所有者が一致すること、P_2 はずっと資源が共に獲得中にならないことを意味する。モデル検査の結果、それぞれ次の反例が出力された。

リスト 5.7: 反例：誤った LTL 式

```
wl206032:1_7 ogawanaoya$ spin -t prsc.pml
Never claim moves to line 94    [(1)]
          (e,e)
                  (e,Q)
                  (Q,Q)
                  (e,Q)
              (P,Q)
Never claim moves to line 93    [((!((pr==sc))&&((pr!=e)&&(sc!=e))))]
                  (P,e)
Never claim moves to line 97    [(1)]
spin: trail ends after 17 steps
#processes: 3
                pr = P
                sc = e
 17:    proc  2 (userQ) prsc.pml:35 (state 25)
 17:    proc  1 (userP) prsc.pml:27 (state 14)
 17:    proc  0 (:init:) prsc.pml:51 (state 12) <valid end state>
 17:    proc  - (never_0) prsc.pml:98 (state 8) <valid end state>
3 processes created
```

リスト 5.8: 反例：正しい LTL 式

```
wl206032:1_7 ogawanaoya$ spin -t prsc.pml
Never claim moves to line 58    [(1)]
          (e,e)
                  (e,Q)
                  (Q,Q)
                  (e,Q)
              (P,Q)
                  (P,e)
                  (P,Q)
Never claim moves to line 57    [(((pr!=e)&&(sc!=e)))]
Never claim moves to line 62    [(((pr!=e)&&(sc!=e)))]
```

```
12 spin: trail ends after 21 steps
13 #processes: 3
14                 pr = P
15                 sc = Q
16  21:    proc  2 (userQ) prsc.pml:37 (state 14)
17  21:    proc  1 (userP) prsc.pml:27 (state 14)
18  21:    proc  0 (:init:) prsc.pml:51 (state 12) <valid end state>
19  21:    proc  - (never_0) prsc.pml:61 (state 9)
20 3 processes created
```

リスト 5.7 は 3 行目から 7 行目が変数 (pr,sc) の遷移である。利用者 Q が両方の資源を獲得したあとプリンタだけを手放して利用者 P がプリンタを獲得したところで反例が出力された。リスト 5.8 は 3 行目から 9 行目が変数 (pr,sc) の遷移である。利用者 Q が一度、両方の資源を獲得したあと利用者 P がプリンタを、利用者 Q がスキャナを獲得した状況で反例が出力された。

5.1.3 考察

時差式信号機の例で『各信号が青⇒黄⇒赤の順に変化する』ことを意味する LTL 式に誤りを混入させた。検出された反例から、正標本集合は正しい前提同士またはその一部分を繋げられる書き方が望ましいことがわかった。

プリンタとスキャナの例でそれぞれの反例を比較すると途中まで同じ遷移であることがわかる。ここでリスト 5.7 の反例の遷移列 $(e,e) \Rightarrow (e,Q) \Rightarrow (Q,Q) \Rightarrow (e,Q) \Rightarrow (P,Q)$ を遷移 A、リスト 5.8 の反例の遷移列 $(e,e) \Rightarrow (e,Q) \Rightarrow (Q,Q) \Rightarrow (e,Q) \Rightarrow (P,Q) \Rightarrow (P,e) \Rightarrow (P,Q)$ を遷移 B とする。遷移 B より、遷移 A の時点ではまだデッドロックは起こっていない。遷移 A が正標本集合に含まれればモデルも誤っているがとりあえず性質 P_1 に誤りがあると特定できる。このように正標本集合も負標本集合もモデル検査で用いるモデルが同じである以上、初期状態から途中までの遷移が同じになる状況は起こりやすい。標本集合を用いて誤りを特定するためには遷移 A と遷移 B が区別できなければならないが、それは真のモデル $\tilde{M}$ の詳細を理解していないと難しい。もはや区別できるということはモデルを忠実に書けることに等しい。よって、正標本集合は初期状態からの遷移で書く必要があるものには適さないと考えられる。

負標本集合の経験から考察する。負標本集合 S_n は真のモデルのコンプリメント $\tilde{M}^c$ を考えると書けた。反例 e が負標本集合に含まれているとすると定義 3.5.2 より $e \in \tilde{M}^c$、すなわち $e \notin \tilde{M}$ となる。しかし定義 3.1.1 より $e \in M$ なのでモデル M に誤りがあると特定できた。同様に真の性質のコンプリメント $\tilde{P}^c$ で考えてみる。すなわち性質 $\tilde{P}$ を満たさない集合 S'_p を書く。定義 3.5.1 より $S_p \subseteq \tilde{P}$ だったので、$S'_p \subseteq \tilde{P}^c$ とする。ここで $S'_p = S^c_p$ を仮定する。反例 e が集合 S'_p に含まれていないとすると定義 3.5.1 より $e \in \tilde{P}$ となる。しかし定義 3.1.1 より $e \notin P$ なので性質 P に誤りがあると特定できる。実際には反例 e が集合 S'_p に含まれていないからといって反例が性質 $\tilde{P}$ を満たすとは言えない。しかし集合 S'_p の要素を増やすことで誤り特定の精度を向上させることができる。モデル M の誤りは反例が

負標本集合 S_n に含まれていることで特定でき、性質 P の誤りは集合 S'_p に含まれていないことで特定できる。

いずれにせよ負標本集合とは違い反例のどの遷移も集合に含まれている、もしくは含まれていないことを示さなければならないことがわかった。

第6章　まとめ

まず、集合論を用いて提案手法を形式的に整理することができた。形式化のところではモデル検査で得られた反例が真の反例、すなわちモデル作成者の意図する検査対象に対応した反例であるかがモチベーションとなっていた。検査対象と実際のモデル検査で用いるものを区別することで提案手法が明確になった。

負標本集合について、構文を定義することで書くことができた。構文は検査対象の振る舞いではないと確信できるものを変数の組の 1 遷移で定義した。1 遷移で定義したことで判定方法も単純な疑似コードで示すことができた。

負標本集合の評価点として書きやすさと検出の十分性を挙げた。書きやすさについて、負標本集合は変数にだけ着目して書ける点、振る舞いの一部分だけで書ける点を示した。検出の十分性については例が不十分であったため評価することができなかった。

正標本集合について、有効な記述方法は見つからなかった。時差式信号機の例から検査対象の性質を満たす振る舞いだと確信できるもの同士を繋げられる書き方ができると望ましいことがわかった。

第7章　今後の課題

負標本集合の評価について、検出の十分性を評価するために多くの例をこなすことが挙げられる。モデル化の誤りの中でも、検出できる誤り、検出できない誤りの分類が見込める。また、定性的な評価しか行なえなかったので定量的な評価も今後の課題となる。

負標本集合の判定方法について、疑似コードの実装が挙げられる。実装することで自動的に判定することができる。

正標本集合について、まずは有効な書き方を見つけることが課題となる。次に判定方法が課題となる。最後に正標本集合の評価を行なう。評価観点は負標本集合と同様に書きやすさと検出できるものと出来ないものを示す検出の十分性が考えられる。

また、反例が標本集合に含まれていないときの対処法を考える必要がある。

参考文献,関連図書

[1] Matthew B. Dwyer, George S Avrunin, James C. Corbett, Property Specification Patterns for Finite-State Verification*, FMSP '98 Proceedings of the second workshop on Formal methods in software practice, 1998.

[2] 吉岡信和, 青木利晃, 田原康之, SPIN による設計モデル検証, 近代科学社, 2008.

[3] 中島震, SPIN モデル検査, 近代科学社, 2008.

[4] Mordechai Ben-Ari, (中島震, 谷津弘一, 野中哲, 足立太郎 訳), SPIN モデル検査入門, オーム社, Vol.17, 2010.