修 士 論 文

モデル検査ツールにより出力された反例に基づく誤り特定に関する研究

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

陳 適

2012 年 6 月

修 士 論 文

モデル検査ツールにより出力された反例に基づく誤り特定に関する研究

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　二木厚吉 教授
審査委員　鈴木正人 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

0910753 陳 適

提出年月: 2012 年 5 月

概 要

モデル検査器より出力される反例の自動解析において、既存研究では誤りの特定までは至っていないが、本研究では、具体的な問題の領域知識に注目し、race conditions という並行計算の重要問題を対象として取り上げ、race conditions 問題の特徴を付加情報とするモデルの誤り自動特定法を提案した．また、提案手法に従って反例解析ツールを実装して、race conditions 問題のモデル例に適用し、提案手法の評価を行った．

目次

第 1 章 はじめに

1.1 背景

ソフトウェアはあらゆる面で人々の生活により深く関わるようになっており，ソフトウェアの安全性と信頼性もますます重視されている．一方，ソフトウェアは大規模化，複雑化の傾向があるため，仕様の正しさを網羅的に検査できる技術として，モデル検査の実用化に注目されている．モデル検査では，モデルが指定性質を満たさない場合，反例（性質を満たさないシステムの実行列）が出力される．出力された反例の解析によって，モデルまだは仕様の問題を発見することができる．

ただし，モデル検査では，モデルの振る舞いが複雑になると，出力される反例は長くなり，読みづらくなるため，手作業で反例を解析してモデルの誤りを特定するには長時間を要する場合もある．モデル開発者の負担を軽減するために，モデル検査器より出力される反例を自動的に解析し，モデルの誤り箇所を自動に特定することが期待される．

1.2 研究の動機

近年，モデル検査の反例解析に関する研究がされているが，既存研究の何れもモデルの反例解析に加工された情報の提供までに止まっていって，誤りの自動特定までは至っていない．そのため，最終的に人間が提供された加工情報と反例を見て，誤り箇所を見つける必要がある．本研究ではモデル検査器より出力される反例に基づいて，モデルの誤り箇所を自動に特定することに挑戦し，誤り自動特定の実現可能性を検証することにした．

1.3 研究の成果

本研究は並行計算に重要な race conditions 問題を対象として取り上げ，race conditions 問題の反例に基づく誤りの自動特定手法を提案した．また，提案手法に従って解析ツールを実装し，race conditions 問題のモデル例に適用した．適用実験の結果，実験に使った何れのモデル例の誤り箇所が全て自動的に特定できた．提案手法は有効であることが分かった．提案手法は反例に基づくモデルの誤り自動特定の試みであり，今後，race conditions 以外にも様々な問題に対して，提案手法のような考え方でモデルの誤り自動特定やモデルの自動修正などの研究が期待される．

1.4 論文の構成

本論文の構成として，まず第 1 章は研究の背景，動機および成果を述べ，第 2 章ではモデル検査について概要的に説明した後, 反例解析及び反例解析の関連研究を紹介し，本研究の特徴を述べる．第 3 章は研究対象となる race conditions 問題とその特徴を取り上げ，第 4 章は反例に基づく race conditions 問題の誤り自動特定および修正方法の提示法を説明する. 第 5 章と第 6 章では提案手法に従って反例解析ツールの実装と適用を述べる．最後に，第 7 章は研究内容のまとめと今後の課題になる．

第 2 章　モデル検査と反例解析

2.1　モデル検査概要

モデル検査とは形式的手法のひとつである．形式的手法では，数学的・論理的基盤に基づいて検査したい性質の正しさを証明する．モデル検査では，ソフトウェアやハードウェアなどを検査対象とした状態遷移モデルを有限オートマトンに対応付け，有向グラフで表現する．特徴は有向グラフの全ての遷移系列を網羅的に全自動探索することである．網羅探索を行うため，通常の試験などでは発見しにくい実行タイミングによって発生する問題の検出に適している．モデルが与えられた性質に満たさない場合，反例という検査しない性質に満たさないシステムの実行列が出力される．反例の解析によってモデルまたは検証対象のソフトウェアやハードウェアの問題が発見できる．

モデル検査の流れは下図に示す．

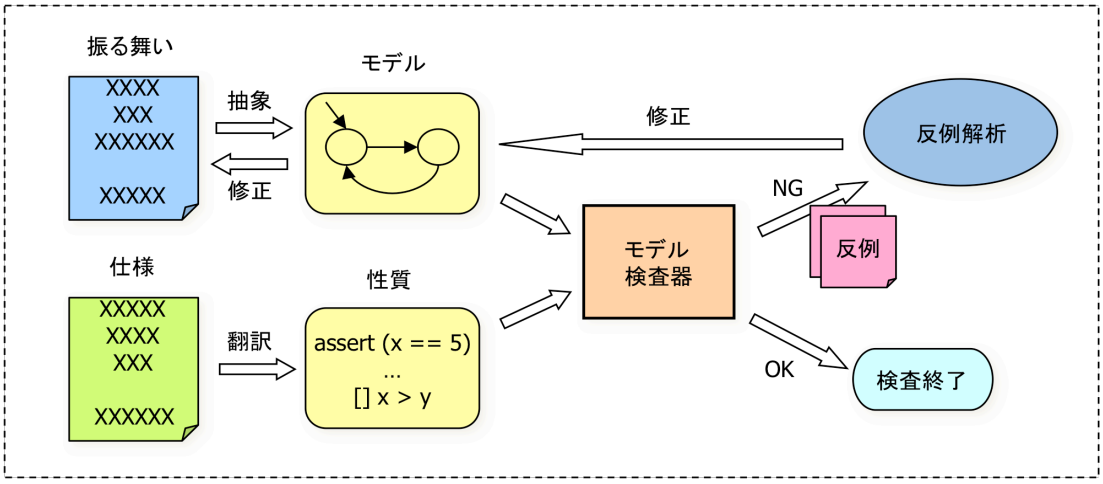


図　1　モデル検査の流れ

モデル検査は以下の流れで実施する．

1）検査モデルの準備
検査したいシステムの振舞いを記述する．設計書や仕様書，ソースコードなどから検査する内容に関連する部分を切り出し，検査に必要な最低限の情報に抽象化して検査モデを作成する．また，検査対象モデルに対して，安全性

(safety), 活性(liveliness) などを検査するために用いる性質を表明や論理式により記述する．

２）モデル検査の実施
用意した検査モデルと性質をモデル検査ツールに投入し，検証を行う．

３）反例解析
性質に満たさない反例が出力された場合，その反例の実行列を分析し，問題となる誤り箇所を見つかり，モデルまたは検査対象の修正を行う． 修正したモデルに対して，また上記の流れで検査を繰り返し実施する．

本研究は上記の反例解析に注目し，モデルの誤りを自動に特定できる反例解析法を提案する．

2.2 モデル検査ツール SPIN

モデル検査ツール SPIN とは，AT&T Bell 研が開発したモデル検査ツールである [7]． SPIN では非決定的な振舞いを専用記述言語 Promela で記述する．Promela での記述を可能な動作を網羅的に探索可能な検査器（C 言語で記述）に変換し，実行できるようにコンパイルする．指定した性質が成立するかどうか自動的にチェックする．検査する性質は，ラベルや表明，性質オートマトンで指定可能である．並行動作や非決定動作を乱数により選択して実行するシミュレーション実行機能がある．また，LTL(Linear Temporal Logic) を性質オートマトンに自動変換する機能も組み込まれている．検査する性質に違反した場合は，反例として違反にいたる経路を示すことができる．Promela 中では状態をラベル，遷移を goto 文で記述するのが一般的である．遷移のガード条件を非決定的に記述可能という特徴がある．

2.3 反例解析とその関連研究

モデルが指定性質を満たさない場合，モデル検査器より反例[5]（性質を満たさないモデルの動作シーケンス）が出力される．反例解析は，反例にあるモデルの動作シーケンスを分析し，モデルが指定性質を満たさない原因を究明することによってモデルまたは仕様にある問題点を発見することである．

反例解析によってモデルまたは仕様の問題点が発見できるが，モデルの振る舞いが複雑になると，手作業で反例を解析することは困難である．反例解析を支援するために，反例の自動解析が研究されている．既存研究の殆どは反例と正例（性質を満たすモデルの動作シーケンス）を比較する方法を取っている．解析に使用する反例と正例の数によって，これらの研究は２種類に分類できる．

分類 1 は複数反例と複数正例を分析し，反例と正例の共通的な差分を抽出して反例解析の材料として提供している．Groce 等は状態遷移，invariant，正例から反例

への変形三つの観点で反例と正例の差異を提示する方法[1]を提案している．Ball 等は，反例にしか存在しない状態遷移に注目する解析方法[2]を提案している．

分類２は正例のうち，反例からの編集距離が最小の正例を探索し，両者の差分を提供している．Groce 等は Pseudo-Boolean Solver で編集距離の最も近い正例を求め，反例と比較する方法[3]を提案している．熊澤等は修正候補となる最も類似正例の求めは有向グラフの最短経路問題に帰着させる方法[4]を提案している．

上記２種類の関連研究の何れも具体的な領域知識を意識していないため，反例と正例の比較結果を提供しているが，結局人間がその比較結果と反例を参照して，誤り箇所を見つける必要がある．

2.4　本研究の概要

本研究はモデル検査器より出力される反例に基づいて，モデルにある誤りを自動に特定し，修正方法を自動に提示する手法を提案する．

現実問題の解決には様々なシステムが作られている．システムの振る舞いを抽象的に表現するモデルも様々な種類がある．それぞれのシステム（モデル）にはそれぞれの領域知識が入っているため，モデルの反例解析を行う際に，これらの領域知識を無視して，汎用的に誤り特定をすることが難しいである．逆に，システム（モデル）の領域知識を意識し，そのシステム（モデル）が発生する問題の特徴を付加情報として反例解析に利用すれば，誤りを自動に特定できる可能性があると考える．

関連研究の何れも領域知識を考慮せずに反例解析手法を提案しているため，誤り箇所の特定までは至っていないが，本研究はある特定種類の問題の特徴に注目し，モデルにある特定種類の問題に関する誤り箇所の特定および修正方法の提示を自動的に行う手法を提案する．提案手法はモデル検査器を改造せず，既存のモデルをそのまま入力として適用できる簡潔な方法である．

提案手法の概念図を以下に示す．

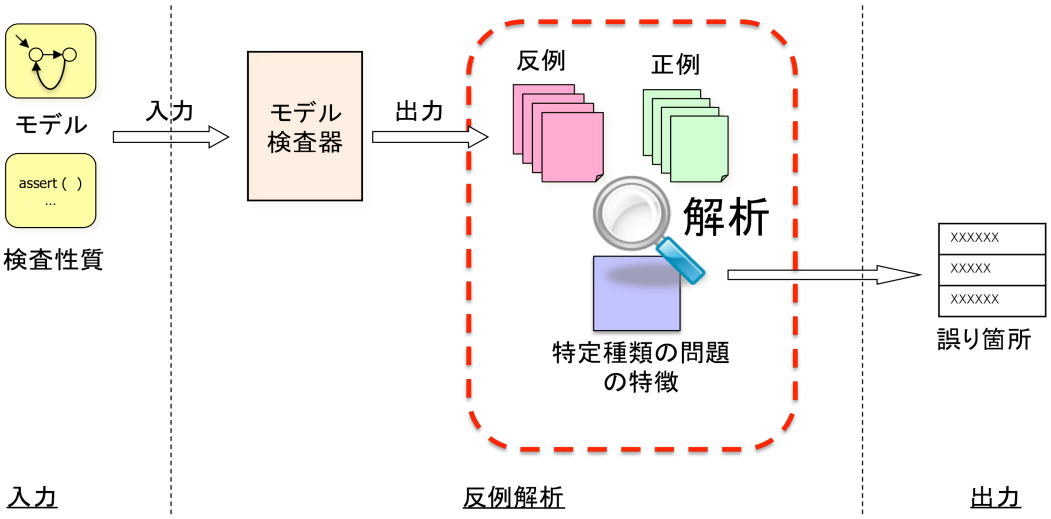


図　2　提案手法の概念図

2.5 本研究の対象

現実システムを表現するモデルの反例に様々な種類の問題があるが, 本研究は並行計算モデルによくある race conditions という問題を扱い対象とし，反例の誤り自動特定を実践する.

本研究の研究対象を以下のように限定する.

- 並行システムの race conditions 問題の反例を解析対象とする.
- 検証する性質を安全性に限定し，性質の記述は表明とする.
- モデル検査器を SPIN とする.

第３章　race conditions 問題

3.1　race conditions 問題とは

race conditions[6]（競合状態）は，処理順序によって処理は予想しない結果になってしまうことをいう．race conditions は設計の不十分な電子工学システム，ソフトウェアでもよく発生する．

並行システムにおいて，複数プロセスの間に共有資源に対する read set(読込み処理)と write set(書込み処理)が空集合でない場合，処理は予想せぬ結果になって race conditions が発生する．race conditions 問題を回避するには，同時にただ 1 つのプロセスのみが実行可能なアトミック区間を設け，読み書きの重なりを無くす必要がある．

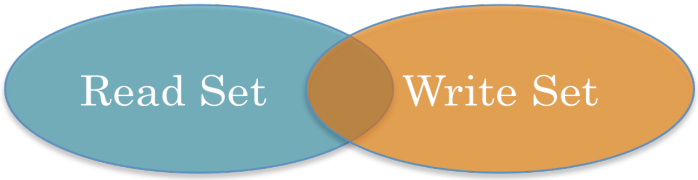


図　3　race conditions 問題

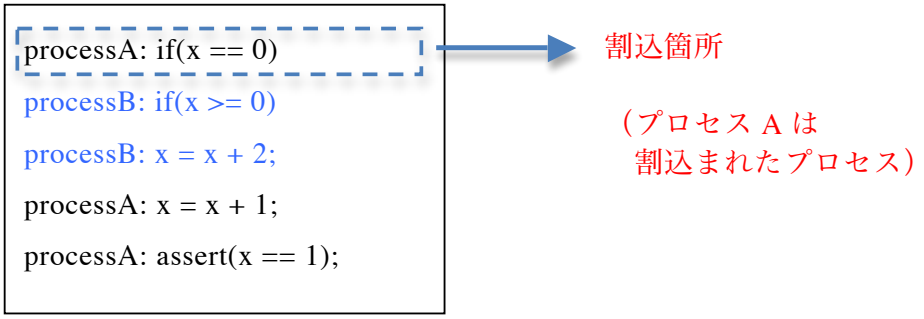
race conditions 問題が起こりうる例を以下に示す．

```
 1 int x = 0;
 2
 3 processA(){
 4     if(x == 0)
 5         x = x + 1;
 6         assert(x == 1);
 7 }
 8
 9 processB(){
10     if(x >= 0)
11         x = x + 2;
12 }
13
14 void main{
15       run processA();
16       run processB();
17 }
```

この例に行われる処理について簡単に説明する．

プロセス A とプロセス B が共有変数である x に対してアクセス処理を行う．二つのプロセスがそれぞれ条件判定をした後に x の値を更新する．プロセス A とプロセス B が main 処理に起動された後，並行で実行するため，二つプロセスの各処理ステップが交替で実行される可能性がある．この例において，race conditions 問題が起こった場合と起こらない場合，処理の実行順序を以下に示す．

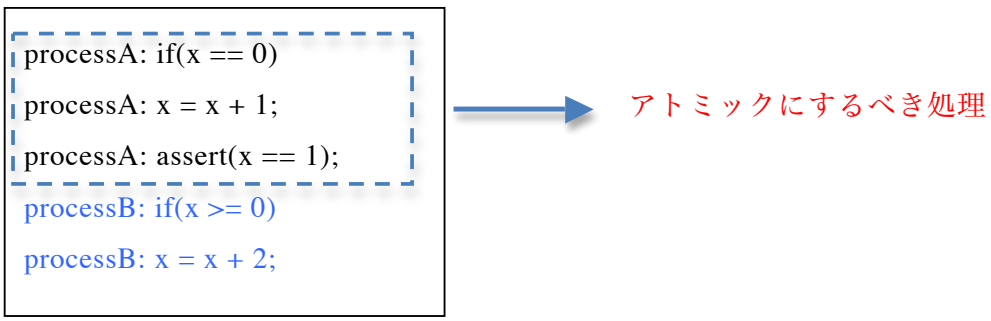
- race conditions 問題が起こった実行例の実行順序



プロセス A が x の値は 0 であることを判定した後，次に x の値を変えようとしている時にプロセス B に割込まれた．そこでプロセス B が x の値を変えてしまったため，プロセス A に x の値は予想通りになれなかった．

本論文ではプロセス B に割込まれる直前のプロセス A の処理ステップを「割込箇所」と呼び，プロセス A を「割込まれたプロセス」と呼ぶ．

- race conditions が起こらなかった実行例の実行順序



上記の実行順序のように割込が発生しなく，プロセス A に x の値を意識している処理が連続で実行できたら，プロセス A にある x の値は予想通りになった．

上記のプロセス A にあるような共有資源の値を参照後，参照値を意識している処理をアトミックに実行されることで race conditions 問題の発生が防ぐことができる．本論文ではこれらの処理を「アトミックにするべき処理」と呼ぶ．

3.2 race conditions 問題の特徴

3.1 節の race conditions 問題が起こった実行例のように，race conditions が発生した場合, あるプロセスの共有資源に対するアクセス処理が他のプロセスに割込まれたことがある．また，共有資源に対するアクセス処理が中断されずに連続で処理することによって，race conditions 問題を回避することができる．本節では race conditions 問題発生時の特徴を明確にする．

定義 1　並行システムにある各処理の種別を次のように定める．
並行システムにある各処理の処理種別 $op \in OP$，ここで，
$OP = \{cond(gv), read(gv), write(gv), assert(lv|gv), other(\varepsilon|lv), |lv \in LV, gv \in GV\}$，$LV$ は並行システムにあるローカル変数の集合, GV は並行システムにあるグロバル変数の集合．*cond(gv)*はグロバル変数に関する条件判定処理，*read(gv)*はグロバル変数に関する読込み処理，*write(gv)*はグロバル変数に関する書込み処理，*assert(lv|gv)*は変数種類に関連しない表明，$other(\varepsilon|lv)$ は変数に関連しない処理，またはローカル変数に関する処理.

定義 2　並行システムの実行列にある各実行ステップを次のように定める．
並行システムの実行ステップ $s_i \in \Sigma$, ここで，$\Sigma = P \times OP \times SRC$．$P$ は並行システムにあるプロセスの集合．SRC は並行システムソース箇所の集合.

定義 3　race conditions 問題発生時の特徴にとして，反例に以下の実行列が含まれる．
$s_1,...,s_l,...,s_m,...,s_n$　ここで,
$s_l = (p_i, cond(v) \mid read(v), src_l) \wedge s_m = (p_j, write(v), src_m)$
$s_n = (..., assert(lv|gv) \Rightarrow false, src_n)$
$l < m < n, i \neq j$
l, m, n, i, j は自然数である．並行システムのプロセス $p_i \in P$，並行システムのソース箇所 $src_i \in SRC$．s_l はプロセス p_i にあるグロバル変数に関する条件判定処理，または読込み処理の実行ステップ．s_m は別のプロセス p_j にあるグロバル変数に関する書込み処理の実行ステップ．s_n は反例の最後にある違反した表明ステップである．s_n は起動プロセスに関連しない．

3.1 節の race conditions 問題が起こった実行例の場合，実行ステップ $s_l = (processA, cond(x), if(x == 0))$, 実行ステップ $s_m = (processB, write(x), x = x + 2)$, 実行ステップ $s_n = (processA, assert(x), assert(x == 1))$.

3.3　race conditions 問題の誤り特定の難しさ

並行システムにおいて，複数プロセス（スレッド）が並行で実行されて各プロセス（スレッド）の実行順序は非決定的であることが多い．そのため，race conditions 問題が発生する場合，誤り箇所が 1 つしか存在しなくでも，複数プロセスの実行順序により反例のバリエーションが多く，人間が反例を見て原因箇所を特定することが困難である．誤り箇所が複数存在する場合，特定はさらに難しくなる．

3.1 節の race conditions 問題が起こりうる例をベースにして，問題箇所数（ここは問題箇所を含むプロセス数にする）と総プロセス数によって，SPIN より出力される反例数を示すためのモデル例を Promela で記述し，SPIN で検証してみた．出力された反例数は下表に示す．

表 1　race conditions 問題の反例のバリエーション

モデル名	問題箇所数	プロセス数	反例数
rc_example1	1	2	1
rc_example2	1	10	19172
rc_example3	2	10	51467

各モデルの Promela 記述及び SPIN の出力は以下に示す．

・モデル rc_example1

```
 1 int x =0;
 2
 3 active[1] proctype A(){
 4     if
 5     ::x == 0 ->
 6         x = x + 1;
 7         assert(x == 1)
 8     fi
 9 }
10
11 active[1] proctype B(){
12     if
13     ::x >= 0 ->
14         x = x + 2
15     fi
16 }
```

・モデル rc_example1 に対する SPIN の出力

```
(Spin Version 6.0.1 -- 16 December 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations +
        acceptance   cycles     - (not selected)
        invalid end states  - (disabled by -E flag)

State-vector 28 byte, depth reached 7, errors: 1
       17 states, stored
        5 states, matched
       22 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

    4.653       memory usage (Mbyte)

unreached in proctype A
        (0 of 6 states)
unreached in proctype B
        (0 of 5 states)
```

・モデル rc_example2

```
 1 int x =0;
 2
 3 active[1] proctype A(){
 4     if
 5     ::x == 0 ->
 6         x = x + 1;
 7         assert(x == 1)
 8     fi
 9 }
10
11 active[9] proctype B(){
12     if
13     ::x >= 0 ->
14         x = x + 2
15     fi
16 }
```

・モデル rc_example2 に対する SPIN の出力

```
(Spin Version 6.0.1 -- 16 December 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations +
        acceptance   cycles     - (not selected)
        invalid end states  - (disabled by -E flag)

State-vector 92 byte, depth reached 31, errors: 19171
   118097 states, stored
   433539 states, matched
   551636 transitions (= stored+matched)
        0 atomic steps
hash conflicts:       26018 (resolved)

Stats on memory usage (in Megabytes):
    13.515      equivalent memory usage for states (stored*(State-vector +
overhead))
     5.370      actual memory usage for states (compression: 39.73%)
                state-vector as stored = 20 byte + 28 byte overhead
     4.000      memory used for hash table (-w19)
     0.458      memory used for DFS stack (-m10000)
     9.731      total actual memory usage

unreached in proctype A
        (0 of 6 states)
unreached in proctype B
        (0 of 5 states)
```

・モデル rc_example3

```
 1 int x =0;
 2
 3 active[2] proctype A(){
 4     if
 5     ::x == 0 ->
 6       x = x + 1;
 7       assert(x == 1)
 8     fi
 9 }
10
11 active[8] proctype B(){
12     if
13     ::x >= 0 ->
14       x = x + 2
15     fi
16 }
```

・モデル rc_example3 に対する SPIN の出力

```
(Spin Version 6.0.1 -- 16 December 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations +
        acceptance   cycles     - (not selected)
        invalid end states  - (disabled by -E flag)

State-vector 92 byte, depth reached 32, errors: 51464
   157461 states, stored
   560897 states, matched
   718358 transitions (= stored+matched)
        0 atomic steps
hash conflicts:       46674 (resolved)

Stats on memory usage (in Megabytes):
    18.020      equivalent memory usage for states (stored*(State-vector +
overhead))
     7.031      actual memory usage for states (compression: 39.02%)
                state-vector as stored = 19 byte + 28 byte overhead
     4.000      memory used for hash table (-w19)
     0.458      memory used for DFS stack (-m10000)
    11.391      total actual memory usage

unreached in proctype A
        (0 of 6 states)
unreached in proctype B
        (0 of 5 states)
```

第４章　反例による race conditions 問題の誤り特定

4.1　提案手法の考え方

3.1 節にある race conditions 問題が起こりうる例のように，race conditions 問題が発生した場合，反例には問題が起きた時モデルの実行列が含まれている．その実行列からどのプロセス（「割込まれたプロセス」）がどこ（「割込箇所」）で他のプロセスの共有資源に対する書込み処理に割り込まれたかを判断できる．正例には race conditions 問題が発生しない実行列が含まれている．その実行列に反例の「割込まれたプロセス」の「割込箇所」で始まる連続処理（「アトミックにするべき処理」）が含まれている．

提案手法のアイデアとして，race conditions 問題という具体的な問題の特徴に注目し，誤り特定と修正方法提示を解けやすい問題に変換する点である．提案手法では従来の「誤り箇所の特定」問題を「モデルにおける不正な割込箇所の特定」問題に変換し，従来の「修正方法の提示」問題を「アトミックすべき範囲の提示」問題に変換する．

「モデルにおける不正な割込箇所の特定」問題について，まず race conditions 問題の反例にある全ての「割込箇所」を誤りの潜在箇所として抽出する．その後，反例と正例の比較を通じて不正な割込箇所を特定する．

「アトミックすべき範囲の提示」問題について，正例に特定できた不正な割込箇所で始まる連続処理から適切な範囲を選定し，アトミックすべき範囲として提示する．

ここの考え方に基づいて，本章後ろの各節にて具体的なやり方を詳しく説明する．

4.2　提案手法の流れ

提案手法は以下の４つのステップで反例解析を行う．

①解析対象モデルの全ての反例と正例を出力させる
②出力された反例と正例から解析に必要な情報を抽出する
③抽出した解析用情報と race conditions 問題の特徴に基づき，誤り特定を行う
④特定結果を纏めて出力する

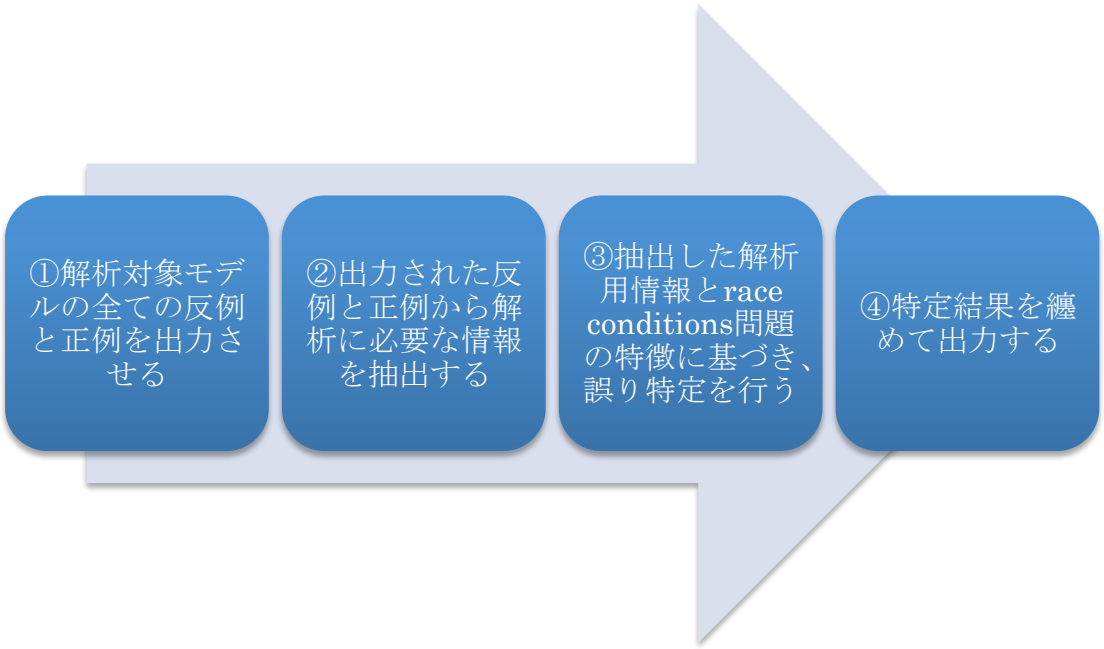


図 4　提案手法の流れ

本章後ろの各節にて上記４つの処理ステップを詳しく述べる.

4.3　反例と正例の生成

4.3.1　反例を全て出力する方法

複数の誤り箇所を一遍に特定するために, 性質を満たさない様々な反例を全て分析する必要がある.

SPIN の場合，反例出力には，まず検証プログラム Pan ファイルの生成が必要である．Pan ファイルは以下のコマンドで生成できる.

```
spin -a [Promela file]
```

モデル検査に検査する性質は大きく分けると安全性(safety)と活性(liveliness)という２種類があるが，本研究は安全性を対象としているため，Pan ファイルをコンパイルする時に，以下ようにコンパイルオプション「-DSAFETY」を指定する.

```
gcc -DSAFETY pan.c -o pan
```

コンパイルした検証プログラムを実行する時に，全ての反例を出力するにランタイムオプション「-e」を使用する必要がある．モデルにループなどによって反例が出力し切れない場合ランタイムオプション「-mN 」を使って指定した深さまで状態区間を探索することができる．また，不要な反例を分析対象外とするために，ランタイムオプション「-E 」を使って終了状態が無効な反例の出力を抑止する．検証プログラムの実行コマンドを以下に示す．

```
pan -mN -e -E
```

3.3 節のモデル rc_example1 の反例出力結果は以下に示す．

```
$ spin -a rc_example1
$ gcc -DSAFETY pan.c -o pan
$ ./pan -e -E
pan:1: assertion violated (x==1) (at depth 5)
pan: wrote rc_example11.trail

(Spin Version 6.0.1 -- 16 December 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states  - (disabled by -E flag)

State-vector 28 byte, depth reached 7, errors: 1
       17 states, stored
        5 states, matched
       22 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

     4.653          memory usage (Mbyte)

unreached in proctype A
        (0 of 6 states)
unreached in proctype B
        (0 of 5 states)

pan: elapsed time 0 seconds
```

4.3.2　正例を全て出力する方法

性質を満たさない様々な反例と比較するために，正例も全て出力する必要がある．SPIN では指定した性質の反例をすべて出力する機能があるが，指定した性質の正

例をすべて出力する機能がない．正例出力するには工夫が必要．

下記は SPIN の検証アルゴリズムを示す図である．モデルのオートマトン S と性質のオートマトン A の積 X が受理できる言語は反例となる．

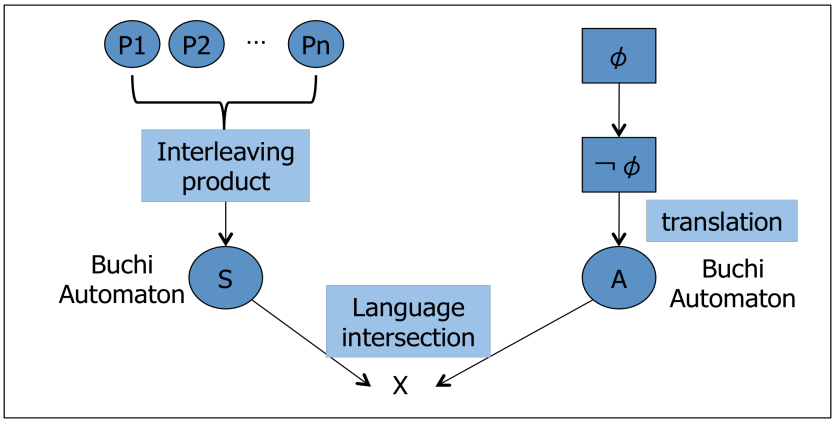


図 5　SPIN の検証アルゴリズム

本研究では，否定した表明で出力した反例を正例とする手法を使う．正例出力の考え方は下図に示す．否定した性質を与えることで，下図のようにモデルのオートマトン S'と性質のオートマトン A の積 X'が受理できる言語は正例となる．

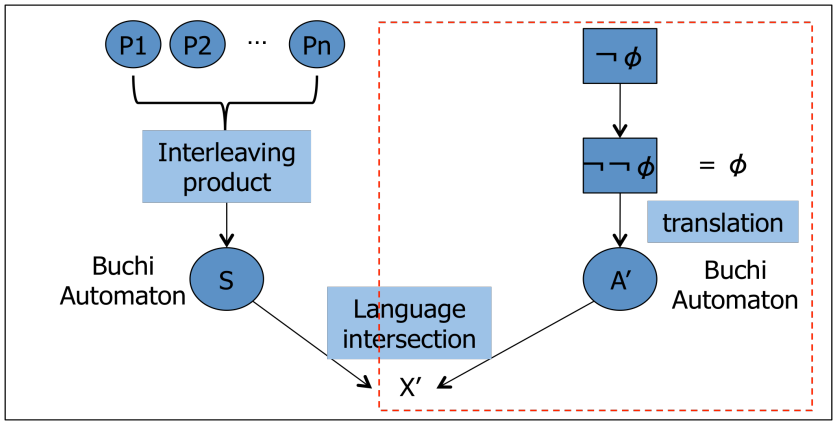


図 6　正例出力の考え方

3.3 節のモデル rc_example1 の正例を出力する場合，モデルにある表明を下記のように編集することになる．

編集前

```
 1 int x =0;
 2
 3 active[1] proctype A(){
 4     if
 5     ::x == 0 ->
 6         x = x + 1;
 7         assert(x == 1)
 8     fi
 9 }
10
11 active[1] proctype B(){
12     if
13     ::x >= 0 ->
14         x = x + 2
15     fi
16 }
```

編集後

```
 1 int x =0;
 2
 3 active[1] proctype A(){
 4     if
 5     ::x == 0 ->
 6         x = x + 1;
 7         assert( !(x == 1) )
 8     fi
 9 }
10
11 active[1] proctype B(){
12     if
13     ::x >= 0 ->
14         x = x + 2
15     fi
16 }
```

編集後のモデルより出力される正例は以下に示す.

```
(Spin Version 6.0.1 -- 16 December 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             - (none specified)
        assertion violations +
        cycle checks            - (disabled by -DSAFETY)
        invalid end states    - (disabled by -E flag)

State-vector 28 byte, depth reached 7, errors: 2
       17 states, stored
        5 states, matched
       22 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

     4.653          memory usage (Mbyte)

unreached in proctype A
        (0 of 6 states)
unreached in proctype B
        (0 of 5 states)

pan: elapsed time 0 seconds
```

4.4 反例と正例から解析用情報の抽出

SPIN モデル検査器から出力される反例は trail ファイルである．trail ファイルに SPIN しか読めないモデルの実行列であり，ファイルには実行スレップのソースや実行時変数の値など人間が読める情報が入っていないため，反例解析を行う前に，trail ファイルから解析に必要な情報を抽出する必要がある．

本研究ではまず trail ファイルを人間が読める実行列に変換させ，そして Race condition 問題の特徴に基づいて共有資源に関する操作を解析用情報として抽出する．

提案手法は以下のコマンドで trail ファイルを実行列に変換する．

```
spin -t[trail file] -p -g -w [Promela file]
```

コマンドに使用するランタイムオプション以下に示す

「-t」　　trail ファイルを読込み，実際の実行列を出力させる
「-p」　　状態や実行ステップのソースを出力させる
「-g」「-w」　グロバル変数とその値を出力させる

上記に得られた実行列の各実行ステップから解析に必要な情報を抽出する．1 行の実行ステップの解析用情報を以下のように定義する．

プロセス ID	処理種別	ソース箇所

- プロセス ID
  実行ステップのプロセス ID である．

- 処理種別
  実行ステップによって各処理を以下のように分類する．
  「cond 処理」（共有資源の条件判定処理）
  「read 処理」（共有資源の読込み処理）
  「write 処理」（共有資源の書込み処理）
  「assert 処理」（性質で記述した表明）
  「other 処理」（共有資源に関連しない処理）

- ソース箇所
  実行ステップにおける Promela のソース箇所を指す．ソース箇所は以下の用に表記する．

Promela ソースファイル名：行数

モデル検査器から出力した 3.3 節のモデル rc_example1 の反例を以下に示す.

```
  1:      proc   1 (B) rc_example1:13 (state 1)   [((x>=0))]
                    x = 0
  2:      proc   0 (A) rc_example1:5 (state 1)    [((x==0))]
                    x = 0
  3:      proc   1 (B) rc_example1:14 (state 2)   [x = (x+2)]
                    x = 2
  4: proc 1 terminates
  5:      proc   0 (A) rc_example1:6 (state 2)    [x = (x+1)]
                    x = 3
spin: rc_example1:7, Error: assertion violated
spin: text of failed assertion: assert((x==1))
  6:      proc   0 (A) rc_example1:7 (state 3)    [assert((x==1))]
                    x = 3
spin: trail ends after 6 steps
```

上記の反例から抽出した解析用情報のイメージを以下に示す.

```
step1-> pid: 1(B), cond(x), rc_example1:13
step2-> pid: 0(A), cond(x), rc_example1:5
step3-> pid: 1(B), write(x), rc_example1:14
step4-> pid: 0(A), write(x), rc_example1:6
step5-> pid: 0(A), assert(x), rc_example1:7
```

4.5 race conditions 問題の誤り特定

本節では, 提案する race conditions 問題の誤り特定, 修正方法提示法を説明する. 説明に使う「cond 処理」,「read 処理」,「write 処理」と「assert 処理」は 4.4 節に定義した処理種別である.

4.5.1 割込箇所の抽出

race conditions 問題で出力される反例に問題原因となる割込処理が存在する. race conditions 問題の誤り特定はそのような割込処理を見つけることである. 提案手法では, まず個々の反例から誤りが潜在している割込箇所を全て抽出し, 一覧にする. その後, 個々の割込箇所が各正例に存在するかどうかを検索し, 正例にも同じ割込が含まれたら, 該当割込箇所を誤り箇所としない, 一覧から除外する.

割込箇所の抽出には, まず割込箇所かどうかの判断が必要. 提案手法では, race

conditions 問題の反例に，他プロセスの「write 処理」に割込まれた「cond 処理」または「read 処理」を割込箇所とする．本来，個々のグロバル変数に対して，モデルの構文解析を行い，グロバル変数ごとに各処理ステップを処理種別で分類することが必要と考えられるが，提案手法では，割込箇所の抽出には，個々のグロバル変数を区別せずに，可能な割込箇所を全て抽出する．その後，反例と正例の比較にて，反例と正例ともに含まれる割込箇所を除外する．

以降，割込箇所の抽出に使用するデータ構造およびアルゴリズムを説明する．

提案手法で抽出した割込箇所を割込箇所情報に格納する．割込箇所情報は以下のように構成される．

割込まれた処理	違反表明	割込距離

- 割込まれた処理
  割込まれた「read 処理」または「cond 処理」の解析用情報
- 違反表明
  反例の終了行にある破れた表明「assert 処理」の解析用情報
- 割込距離
  実行ステップに割込まれた「read 処理」または「cond 処理」から他プロセスの「write 処理」まで同プロセスが継続で実行されたステップ数

実行ステップの解析用情報の定義は以下に示す．

```
//実行ステップの解析用情報
AnalysisInfo{
  pid,  //プロセス ID
  type,  //処理種別
  step  //ソース箇所
}
```

反例，正例の解析用情報の定義は以下に示す．

```
//反例、正例の解析用情報
TrailInfo{
  InfoList<AnalysisInfo>,  //実行列の解析用情報
  trail,  //trail ファイルのプルパス
  getAssert(), //実行列最後の表明行を取得する
  getContinuousSteps(AnalysisInfo)//  指定した実行ステップから始まる
                                  // 同プロセスの連続処理ステップ数を取得する
}
```

割込箇所情報の定義は以下に示す.

```
//割込箇所情報
InterruptInfo {
   stepInfo,  //割込まれた処理
   assertion,  //違反表明
   cSteps   //割込距離
}
```

割込箇所の抽出アルゴリズムを以下に示す.

```
//反例より反例解析用情報一覧を抽出する
List<TrailInfo> counterexample_list = parseCounterExample()
//割込箇所情報を格納する一覧
List<InterruptInfo> interrupt_list

for each TrailInfo ce in counterexample_list
   for each AnalysisInfo line1 in ce
      if line1.type is COND or READ then
         for each AnalysisInfo line2 after line1 in ce
            if line2.pid != line1.pid && line2.type = WRITE then
             interrupt_list.put(InterruptInfo(line1,ce.geAssert(),
                                          ce.getContinuousSteps(line1))
               break
            endif
      endif
```

上記アルゴリズムは以下に説明する.

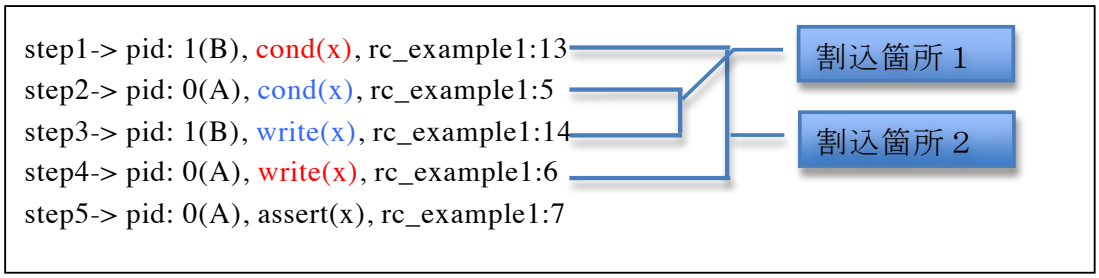
- 個々の反例情報に対して，一行ずつ判定を行う.
- もし判定対象行の処理種別が「read 処理」または「cond 処理」の場合，その行の後ろに他のプロセスの「write 処理」があるかを探索する.
- 他のプロセスの「write 処理」を見つけた場合，その「read 処理」または「cond 処理」，違反表明と割込距離を割込箇所情報として割込箇所一覧に入れる.

注意されるべきこととして，他プロセスの「write 処理」は必ず割込箇所とされる「read 処理」と「cond 処理」の直後に付いているに限らない.「read 処理」と「cond 処理」の後，同プロセスにおけるその他共有変数へのアクセスや，ローカル変数へのアクセスなどの同プロセスの処理が続いていることも多い.割込箇所の後，他プ

ロセスに割込まれるまで同プロセスの実行ステップ数を割込距離とする．以降の誤り特定に割込距離を正例に割込箇所で始まる連続処理ステップ数と比較するため，割込距離を割込箇所情報に格納しておく必要がある．

また，同じ割込箇所が複数の反例に含まれることがあるため，効率よく解析を行うために，同じ割込箇所を 1 つに纏めて割込箇所一覧に入れる．

3.3 節のモデル rc_example1 の反例解析用情報に対して，割込箇所の抽出を以下に示す．



上記の反例より 2 つの割込箇所が抽出される．割込箇所 1 について，プロセス A の「cond 処理」の直後に，プロセス B の「write 処理」に割込まれて，割込距離は 0 で．割込箇所 2 について，プロセス B の「cond 処理」後，プロセス A の「write 処理」までには，プロセス B の step3 の処理があるため，割込距離を 1 とする．

4.5.2　誤り箇所の判定

抽出した割込箇所情報一覧にある個々の割込箇所情報に対して，その割込箇所が誤り箇所かどうかを以下のように正例を参照して判断する．

- 全ての正例に同じ割込が存在しない場合，該当割込箇所を誤り箇所とし，誤り箇所情報一覧に入れる．
- 正例に同じ割込が存在する場合，該当割込箇所を誤り箇所としない．

以降，誤り箇所の特定に使用するデータ構造およびアルゴリズムを説明する．
誤り箇所情報一覧に入れる誤り箇所情報が以下のように構成される．

割込箇所情報	正例の連続実行ステップ数

- 割込箇所情報
  誤り箇所として判断される割込箇所の関連情報
- 正例の連続実行ステップ数
  正例に割込箇所情報の割込まれた処理で始まる同プロセスの連続実行ステップ数

誤り箇所情報の定義を以下に示す.

```
//誤り箇所情報
ErrorInfo {
 interruptInfo,  //割込箇所情報
 wSteps   //正例の連続実行ステップ数
}
```

誤り箇所の判定アルゴリズムを以下に示す.

```
//正例より正例解析用情報一覧を抽出する
List<TrailInfo> witness_list = parseWitness()
//誤り箇所情報を格納する一覧
List<ErrorInfo> error_list

for each InterruptInfo irInfo in interrupt_list
   for each TrailInfo wtInfo in witness_list
      if(wtInfo.getAssert() != irInfo.assertion) then
         continue
      endif
      continuousSteps = wnInfo.getContinueSteps(irInfo.step)
      if(continuousSteps > irInfo.cSteps) then
           error_list.put(ResultInfo(irInfo, continuousSteps))
      else
           if error_list does not contains irInfo then
              break
           else
              remove all items contains irInfo
           endif
      endif
```

上記アルゴリズムは以下に説明する.

- 抽出した割込箇所情報一覧にある個々の割込箇所情報に対して処理を行う.
- 対象割込箇所情報に対応する反例の実行列の最終行は正例実行列の最終行と同じ表明である（プロセス ID, 処理種別, ソース箇所が全て一致する）場合, その正例を比較対象とし，次の処理を行う.

- 比較対象の正例に割込箇所情報の割込まれた処理で始まる同プロセスの連続実行ステップ数を取得し，割込箇所情報の割込距離と比較する．比較結果によって，次の処理を行う．
    - 正例の連続実行ステップ数が割込箇所情報の割込距離より大きい場合，その割込箇所情報を正例の連続実行ステップ数と一緒にセットで誤り箇所一覧に格納する．
    - 正例の連続実行ステップ数は割込箇所情報の割込距離以下の場合，対象割込箇所を誤り箇所としない，割込箇所情報が誤り箇所一覧に格納されているかによって，次の処理を行う．
        - 該当割込箇所情報が誤り箇所一覧に格納されていない場合, 次の割込箇所を処理する．
        - 他の正例との比較で該当割込箇所情報が既に誤り箇所として一覧に格納されている場合, 誤り箇所一覧から該当割込箇所情報を削除して，次の割込箇所を処理する．

3.3 節のモデル rc_example1 により出力された２つの正例から抽出した解析用情報を以下に示す．

正例 1 の解析用情報

```
step1-> pid: 1(B), cond(x), rc_example1:13
step2-> pid: 0(A), cond(x), rc_example1:5
step3-> pid: 0(A), write(x), rc_example1:6
step4-> pid: 0(A), assert(x), rc_example1:7
```

正例２の解析用情報

```
step1-> pid: 0(A), cond(x), rc_example1:5
step2-> pid: 0(A), write(x), rc_example1:6
step3-> pid: 0(A), assert(x), rc_example1:7
```

4.5.1 節で見つけたモデル rc_example1 の反例の割込箇所に対して，上記の誤り箇所の特定アルゴリズムで分析する．二つの正例に割込箇所１（pid: 0(A), cond(x), rc_example1:5）で始まる同プロセス A の連続処理ステップ数は同じく３である．この数字は反例にある割込箇所 1 の割込距離の０より大きいので, 割込箇所 1 を誤り箇所として, 誤り箇所一覧に格納する. 二つの正例に割込箇所２（pid: 1(B), cond(x),

rc_example1:13）で始まる同プロセスの連続処理ステップ数は 1 である．この数字は反例にある割込箇所 1 の割込距離の 1 と同じであるため，割込箇所 2 を誤り箇所としない．

4.5.3 修正方法の選定

並行システムに各プロセスの処理をプロセス単位で全てアトミック処理にすれば，race conditions 問題が発生しないが，システムの処理は並行に実行できなくなる．race conditions 問題を解決するために，適切なアトミック処理範囲を選定する必要がある．race conditions 問題が発生しない限り，アトミック処理範囲を狭くすればするほど，並行システムの実行効率が上がることになる．

4.5.2 節にて誤り箇所と判断された割込箇所ごとに正例を参照して，正例に割込箇所で始まる同プロセスの連続実行ステップ数を割込箇所と一緒に誤り箇所情報一覧に入れた．複数正例に同じ割込箇所が含まれ，それぞれの正例に割込箇所で始まる同プロセスの連続実行ステップ数が実行タイミングによって違う可能性がある．その場合，誤り箇所情報一覧に同じ誤り箇所に対して，複数項目が格納されるため，誤り箇所単位に纏める必要がある．4.5.2 節の誤り箇所の特定では誤り箇所以外の割込箇所を除外してあるため，実行効率の観点で，誤り箇所単位の纏めは以下のように行う．

- 正例に誤り箇所とされた割込箇所で始まる同プロセスの連続処理のうち，最短の連続処理を選択する．

SPIN で生成した検証プログラムが実行する時に，モデルにある各プロセスの起動は非決定的であるため，プロセスの起動順番によって，プロセス ID が変わってくる．割込箇所にある割込まれた処理にはプロセス ID が含まれているため，割込まれた処理のソース箇所が同じとしてもプロセス ID が違うなら，別々の割込箇所とされることになる．

これまでは誤り箇所と判断された割込箇所ごとに特定結果を纏めたが，最終的に修正方法はソース箇所で提示する必要があるため，特定結果をソース箇所ごとに纏める必要もある．同じソース箇所に対して，どのプロセス起動順序でも race conditions 問題を発生させないため，ソース箇所単位の纏めは以下のように行う．

- 誤り箇所情報一覧にあるソース箇所が同じ割込箇所のうち，正例の連続実行ステップ数が大きい方を選択し，1 つに纏め，最終の特定結果とする．

修正方法の選定アルゴリズムを以下に示す.

```
//割込箇所単位特定結果
Map<InterruptInfo, ErrorInfo> trail_result
//ソース箇所単位特定結果
Map<step, ErrorInfo> step_result

//割込箇所単位の特定結果纏め
for each ErrorInfo errInfo in error_list
   InterruptInfo itInfo = errInfo.interruptInfo
   if(!trail_result.contains(itInfo) then
      trail_result.put(itInfo, errInfo)
   else
      if errInfo.wSteps < trail_result.get(itInfo).wSteps then
         trail_result.put(errInfo.interruptInfo, errInfo)
      endif
   endif

//ソース箇所単位の特定結果纏め
for each ErrorInfo errInfo in step_result
   Step itStep = errInfo.interruptInfo.stepInfo.step
   if(!step_result.contains(itStep) then
      step_result.put(itStep, errInfo)
   else
      if errInfo.wSteps > trail_result.get(itStep).wSteps then
         step_result.put(itStep, errInfo)
      endif
   endif
```

上記アルゴリズムは以下に説明する.

- 誤り箇所一覧にある個々の誤り箇所情報を割込箇所単位で以下のように纏める．該当誤り箇所情報の割込箇所情報を抽出し，その割込箇所をキーとしている誤り箇所が割込箇所単位特定結果一覧に格納されているかによって，次の処理を行う.
    - 該当誤り箇所情報の割込箇所情報をキーとしている誤り箇所が割込箇所単位特定結果に格納されていない場合，該当誤り箇所情報を割込箇所単位特定結果に入れる.
    - 該当誤り箇所情報の割込箇所情報をキーとしている誤り箇所が既に割込箇所単位特定結果に格納されている場合，結果に格納されている誤り箇

所情報にある正例の連続実行ステップ数を該当誤り箇所情報にある正例の連続実行ステップ数と比較し，正例の連続実行ステップ数が小さい方に纏める．

- 次に，割込箇所単位特定結果一覧にある個々の誤り箇所情報をソース箇所単位で以下のように纏める．該当割込箇所情報にある割込処理のソース箇所を抽出し，そのソース箇所をキーとしている誤り箇所がソース箇所単位特定結果に格納されているかによって，次の処理を行う．
  - ソース箇所をキーとしている誤り箇所がソース箇所単位特定結果一覧に格納されていない場合，該当誤り箇所情報をソース箇所単位特定結果一覧に入れる．
  - ソース箇所をキーとしている誤り箇所がソース箇所単位特定結果一覧に格納されている場合，一覧に格納されている誤り箇所情報にある正例の連続実行ステップ数を該当誤り箇所情報にある正例の連続実行ステップ数と比較し，正例の連続実行ステップ数が大きい方に纏める．

割込箇所単位の誤り箇所纏めについて，以下のモデル例 rc_example4 を使って説明する．

```
 1 int x = 5;
 2
 3 active[1] proctype threadA()
 4 {
 5     int y, z = 0;
 6
 7     if
 8     ::x==5
 9         ->
10               y = x * 2;
11               z = x + y;
12               assert(y == 10)
13     fi
14 }
15
16 active[1] proctype threadB()
17 {
18     x = 4;
19 }
```

上記モデルより出力される反例が 1 つのみとなり，その反例の解析用情報を以下に示す．

```
step1-> pid: 0(threadA), cond(x), rc_example4:8
step2-> pid: 1(threadB), write(x), rc_example4:18
step3-> pid: 0(threadA), other(), rc_example4:10
step4-> pid: 0(threadA), other(), rc_example4:11
step5-> pid: 0(threadA), assert(), rc_example4:12
```

上記の反例より threadA の「cond 処理」が threadB の「write 処理」に割込まれたことが分かる．

上記モデルより 2 つの正例が出力され，正例の解析用情報以下に示す．

正例 1

```
step1-> pid: 0(threadA), cond(x), rc_example4:8
step2-> pid: 0(threadA), other(), rc_example4:10
step3-> pid: 1(threadB), write(x), rc_example4:18
step4-> pid: 0(threadA), other(), rc_example4:11
step5-> pid: 0(threadA), assert(), rc_example4:12
```

正例 2

```
step1-> pid: 0(threadA), cond(x), rc_example4:8
step2-> pid: 0(threadA), other(), rc_example4:10
step3-> pid: 0(threadA), other(), rc_example4:11
step4-> pid: 0(threadA), assert(), rc_example4:12
```

上記の正例に割込まれた処理 rc_example4:8 に対応する同プロセスの連続処理について，正例 1 では rc_example4:8～rc_example4:10 の 2 ステップに対して，正例 2 は rc_example4:8～rc_example4:12 の 4 ステップになる．割込箇所単位の誤り箇所纏めアルゴリズムでは，同じ割込み箇所の場合，最短の連続処理を選択するため，誤り箇所纏めの結果，rc_example4:8～rc_example4:10 の 2 ステップをアトミック処理にすることは修正方法になる．本モデル例にある性質はローカル変数 y に関する表明であり，rc_example4:11 以降変数 z に関する処理が threadB に割込まれるかどうかは性質に影響しないため，rc_example4:8～rc_example4:10 をアトミックにする修正方法が適切である．

4.6 解析結果の出力

本節は解析結果の出力形式について説明する.

提案手法で特定できた誤り箇所およびその修正方法を解析結果として出力する. 解析結果は以下の形式で出力する.

【誤り箇所 1】の関連情報
・・・
【誤り箇所 n】の関連情報

個々の誤り箇所が以下の関連情報より構成される.

- アトミックにするべきソース範囲

  race conditions 問題の修正方法として下記の形式で出力される.

形式	アトミック処理開始箇所 ～ アトミック処理終了箇所
例	model.pml:10 行目～model.pml:12 行目

- 正しく実行された正例

  正しい実行順序が含まれた正例情報ファイルを参照情報として以下の形式で出力される.

形式	[正しい実行例のフルパス]
例	/Users/chinteki/spin/env/rw_dir/OK/model.trail.txt

解析結果の出力例を以下に示す.

```
誤り箇所特定の結果：

【誤り箇所 1】
次の 4 ステップがアトミック処理になっていない:
[rc_me:5 行目～rc_me:8 行目]
正しい実行例: /Users/chinteki/spin/env/rc_me_dir/OK/rc_me7.trail.txt

【誤り箇所 2】
次の 2 ステップがアトミック処理になっていない:
[rc_me:22 行目～rc_me:23 行目]
正しい実行例: /Users/chinteki/spin/env/rc_me_dir/OK/rc_me2.trail.txt
```

第５章　解析ツールの実装

5.1　解析ツールの構成

提案手法に従って，指定モデルより反例，正例の生成から修正方法の提示まで自動的に行うツールを実装した．解析ツールは以下の二つのツールより構成される．

１）反例・正例生成ツール
　与えられたモデルにおける反例と正例を全て出力するツール

２）誤り特定ツール
　与えられたモデル及びそのモデルの反例と正例をもとに，提案手法に従って，race condition 問題を特定し，修正方法を自動的に提示するツール

解析ツールの構成および構成される各ツールの機能を下図に示す．

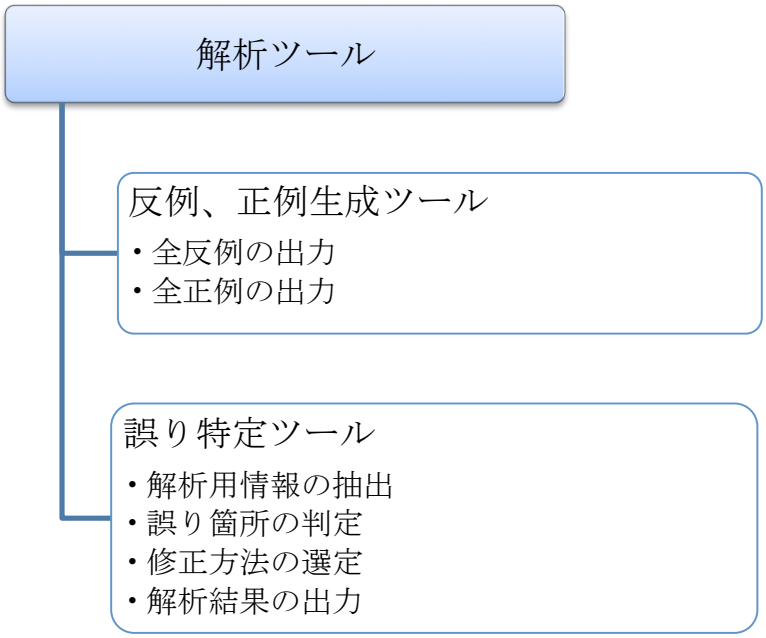


図 7　解析ツールの構成

5.1.1　反例・正例生成ツールの実装

反例・正例生成ツールは以下の機能より構成される.

・入力引数のチェック
・全ての反例ファイルを出力する機能
・モデルにある表明を否定して，全て正例ファイルを出力機能
・出力された trail ファイルから実行列ファイルを出力する機能

反例・正例生成ツールは以下の処理フローにように実装された.

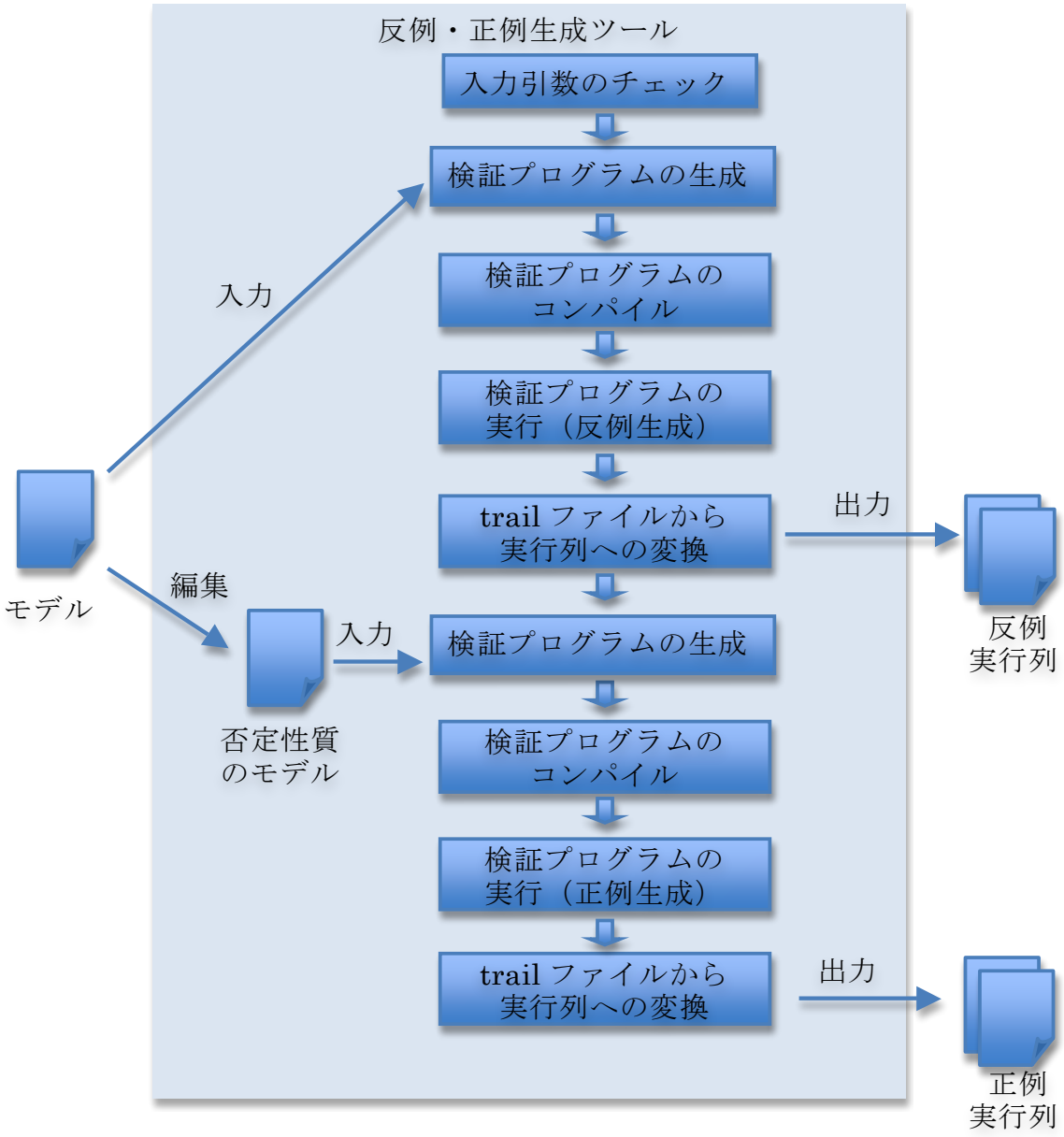


図 8　反例・正例生成ツールの処理フロー

5.1.2　誤り特定ツールの実装

誤り特定ツールは以下の機能より構成される.

- 入力引数のチェック
- 解析用情報抽出機能
- race conditions 問題の誤り特定機能
- race conditions 問題の修正方法提示機能

誤り特定ツールは以下の処理フローに従って実装された.

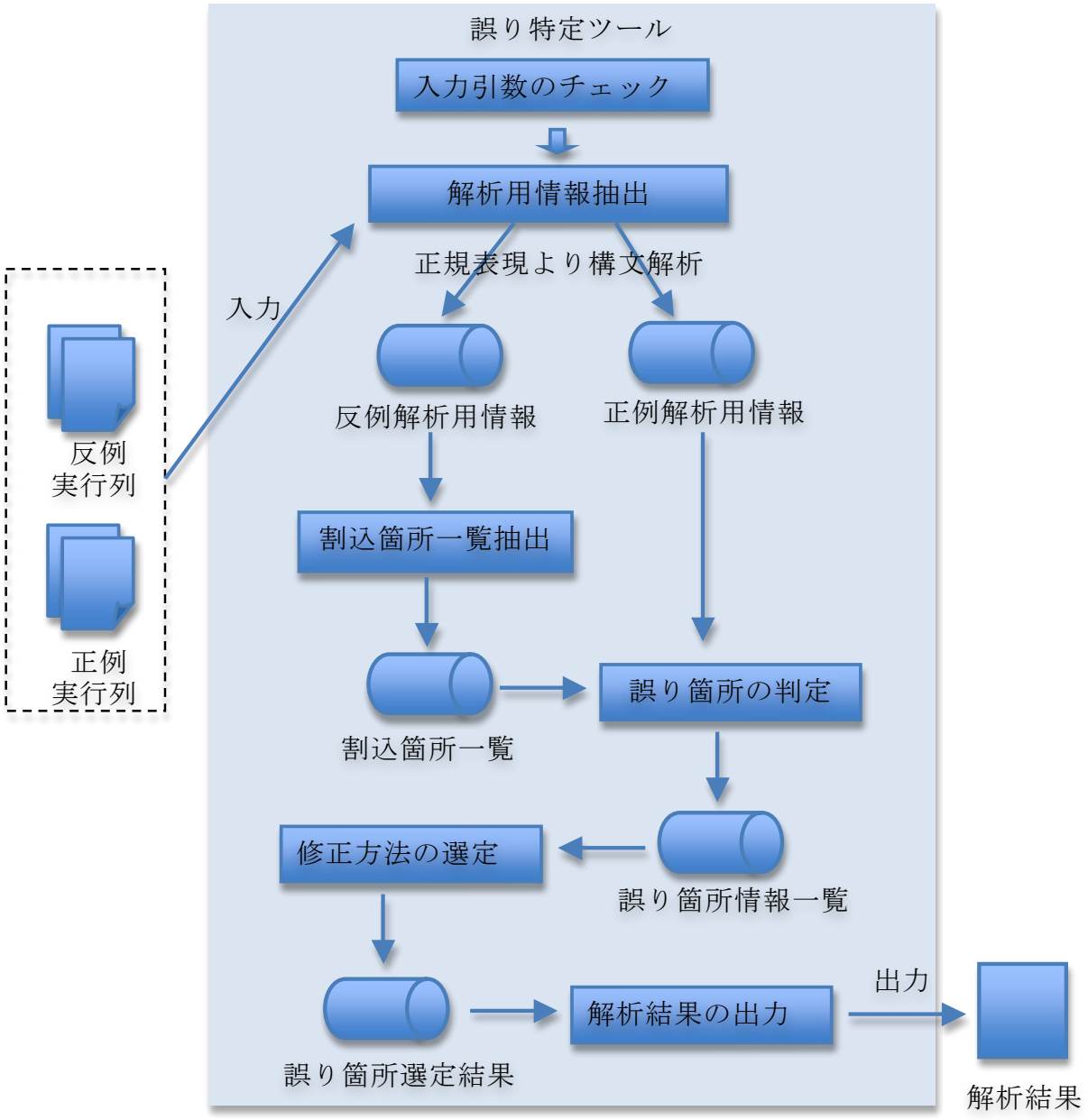


図 9　誤り特定ツールの処理フロー

5.2 解析ツールの実行方法

この節では，反例解析に使用する解析ツールの実行方法を述べる．

解析ツールを実行する際に，反例・正例生成ツール，誤り特定ツールおよび解析対象モデルが同じフォルダに置かれることを想定している．

反例解析するに当たって，解析用情報を得るために，まず反例・正例生成ツールを使って，解析対象モデルの全ての反例と正例を出力する．

反例・正例生成ツールは以下のように実行する．

```
$ ce Promela_file
```

引数の Promela_file は Promela で記述解析対象モデルである．

反例・正例生成ツールとモデル例 rc_lp がフォルダ/User/chinteki/spin/env/ に置かれた場合，反例・正例生成ツールの実行例を以下に示す．

```
$ ce rc_lp

反例出力の結果：
【出力数】767
【出力先】/Users/chinteki/spin/env/rc_lp_dir/NG/
正例出力の結果：
【出力数】768
【出力先】/Users/chinteki/spin/env/rc_lp_dir/OK/

出力所要時間：35.000 秒
```

上記の実行結果に実際に出力した反例と正例数および出力されたそれぞれの反例と正例の実行列の置き場所が提示されている．誤り特定ツールが実行される時にこれらの実行列を参照する．最後に反例と正例の出力に使用した時間が提示されている．

反例と正例の実行列が出力されたら，誤り特定ツールを使って，出力された反例と正例に基づいて誤りを特定する．

誤り特定ツールは反例・正例生成ツールがは以下のように実行する．

```
$ java CEAnalysis Promela_file [search_depth]
```

引数の Promela_file は Promela で記述解析対象モデルである．
引数の search_depth は状態空間の探索深さである．必須ではなく，出力反例の数を減らしたいなどチューニング時のみ使用する．

誤り特定ツールとモデル例 rc_lp がフォルダ/User/chinteki/spin/env/に置かれた場合，誤り特定ツールの実行例は以下に示す．

```
$ java CEAnalysis rc_lp

誤り箇所特定の結果：
【誤り箇所 1】
次の 2 ステップがアトミック処理になっていない:
[rc_lp:7 行目～rc_lp:9 行目]
正しい実行例: /Users/chinteki/spin/env/rc_lp_dir/OK/rc_lp1.trail.txt

解析所要時間 6.226 秒
```

上記の実行結果に誤り箇所が 1 箇所あると提示され，[rc_lp:7 行目～rc_lp:9 行目]の２ステップがアトミックに処理されるべきと示唆している．
また，この２ステップが連続で実行された正例の実行列も参照情報として一緒に提示されている．

第６章　提案手法の適用実験

提案手法を評価するために，解析ツールを用いて評価用のモデルに対して，適用実験を行った．主な評価内容としては，提案手法の有効性と提案手法の性能の２点である．

提案手法の有効性

提案手法の有効性を評価するために，以下の２種類の評価モデルを用意した．

- 評価用モデル

  race conditions 問題が存在すれば，モデルの書き方やモデルの構造に依存せず，提案手法が全て適用できることを評価するため，race conditions 問題が発生する代表的なモデル構造を持つ評価モデルを自作した．

- 典型的なモデル

  race conditions 問題おける典型的なアルゴリズムを実装したモデル例に対して提案手法が適用できることを評価する．

提案手法の有効性を評価するために，以下の手順で適用実験を行った．

- 用意したモデルに race conditions 問題が発生されるために，バグを埋め込む．
- 解析ツールを使用し，上記のバグが埋め込んだモデルに対して適用する．
- 解析ツールの出力（誤り箇所と修正方法）に従って，モデルを修正して，モデルが修正されたかどうかを確認する．

提案手法の性能

用意した評価用モデル例と典型的なモデル例にプロセス数多いモデルや反例と正例がたくさん出力するモデルも含まれているため，提案手法の性能評価が行うことが可能である．

提案手法の性能を評価するため，評価に使用する各モデルに対して，各モデルにあるプロセス数，各モデルより出力される反例数，正例数および誤り箇所数を計っ

た上，反例と正例を出力に使用した時間と誤り特定に掛かった時間を計測した．

適用実験に使用したモデル及び実験結果は下表に示す．

表 2　適用実験の対象と実験結果

分類	モデル	プロセス数	出力反例数	出力正例数	出力時間	誤り箇所数	誤り特定時間
評価用モデル	グロバル変数をサブ関数に操作例	2	4	9	1.5 秒	1	0.3 秒
	表明に複数変数が含まれる例	3	3	7	1.2 秒	1	0.2 秒
	誤り箇所が複数ある例	3	10	8	2.3 秒	2	0.4 秒
	表明にローカル変数が含む例	3	4	1	1.6 秒	1	0.2 秒
	表明はモニターで定義する例	3	12	20	1.8 秒	2	0.5 秒
	プロセス数多い例	10	767	768	35 秒	1	6.2 秒
典型的なモデル	変数同時更新問題	2	1	26	1.8 秒	2	0.7 秒
	読み手書き手問題	3	2633	732	84.5 秒	3	18.3 秒

- 実験環境（CPU:2.26 GHZ Intel Core 2 Duo, Memory: 2GB）

後ろの各節にて適用試験に使用したモデルの適用結果及び解析ツールが提示した修正方法の確認について，詳しく述べる．

6.1　評価用モデル例への適用

本節では，評価用モデル例に対して解析ツールの適用結果を述べる

6.1.1　グロバル変数をサブ関数に操作するモデル例への適用

評価に使用したモデル「rc_sf」は以下に示す．

```
 1 #define MAX 2
 2 #define semaphore byte
 3
 4 byte cnt = 0;
 5 semaphore s=1;
 6
 7 inline up(x){
 8     x++;
 9 }
10
11 inline down(x){
12     if
13     ::(x > 0) ->
14       x --;
15     fi
16 }
17
18 active [2] proctype semademo(){
19     do
20     ::down(s);
21         cnt++;
22         assert(cnt == 1);
23         cnt--;
24         up(s);
25     od
26 }
```

「評価例の説明」

この評価例はセマフォをデモする例である．２２行目では同時に１つのプロセスしか入られない性質が表明で書かれている．グルバル変数 cnt に対する操作が直接ではなく，サブ関数の down()と up()にて行われる．

「実験の目的」

共有変数へのアクセスは直接ではなく，サブ関数経由でアクセスしていることもよくモデルに実装される．解析ツールではそのようなモデルに対しても適用できることを評価する．

反例・正例出力ツールの実行結果は以下に示す．

```
$ ce rc_sf

反例出力の結果：
【出力数】4
【出力先】/Users/chinteki/spin/env/rc_sf_dir/NG/
正例出力の結果：
【出力数】9
【出力先】/Users/chinteki/spin/env/rc_sf_dir/OK/

出力所要時間：1.476 秒
```

誤り特定ツールの適用結果は以下に示す.

```
$ java CEAnalysis rc_sf

誤り箇所特定の結果：
【誤り箇所 1】
次の 4 ステップがアトミック処理になっていない:
[rc_sf:13 行目～rc_sf:22 行目]
正しい実行例: /Users/chinteki/spin/env/rc_sf_dir/OK/rc_sf8.trail.txt

解析所要時間 0.295 秒
```

提示されたアトミックにするべき範囲が[rc_sf:13 行目～rc_sf:22 行目]となっているが，rc_sf の 13 行目はサブ関数 down()の処理開始であるため，モデルを修正する時に，[rc_sf:20 行目～rc_sf:22 行目]をアトミック処理にする.

誤り特定ツールが提示した修正方法でモデルを下記のように修正して, 修正後のモデルは反例・正例出力ツールで修正したかどうかを確認する.

```
 1 #define MAX 2
 2 #define semaphore byte
 3
 4 byte cnt = 0;
 5 semaphore s=1;
 6
 7 inline up(x){
 8     x++;
 9 }
10
```

```
11 inline down(x){
12     if
13     ::(x > 0) ->
14       x --;
15     fi
16 }
17
18 active [2] proctype semademo(){
19     do
20     ::d_step{ down(s);
21         cnt++;
22         assert(cnt == 1);}
23         cnt--;
24         up(s);
25     od
26 }
```

反例・正例出力ツールの実行結果は以下に示す.

```
$ ce rc_sf_fix

指定したモデルから反例が出力されませんでした.
```

適用実験の結果，解析ツールがモデルの誤り箇所が正しく特定できた．解析ツールはグロバル変数をサブ関数に操作するモデルに適用できることが分かった.

6.1.2　表明に複数変数が含まれるモデル例への適用

評価に使用したモデル「rc_mv」は以下に示す.

```
 1 int x, y = 0;
 2
 3 proctype A(){
 4   if
 5     ::x == 0 && y == 0 ->
 6         x = x + 10;
 7         y = y + 10;
 8         assert(x == 10 && y == 10)
 9     fi;
10
11 }
12
```

```
13 proctype B(){
14     if
15     ::d_step{x == 0 ->
16         x = x + 10}
17     fi;
18 }
19
20 proctype C(){
21     if
22     ::d_step{y == 0 ->
23         y = y + 10;
24    }
25     fi;
26 }
27
28 init{
29       run A();
30       run B();
31       run C()
32 }
33
```

「評価例の説明」

この評価例は表明に複数変数が含まれるモデル例である．プロセス A にグルバル変数 x とグロバル y が含まれる表明がある．グロバル変数 x の書込み操作を持つプロセス B とグロバル変数 y の書込み操作を持つプロセス C のどちらにも割込まれる可能性がある．race conditions 問題をさせないために，プロセス B とプロセス C の両方の割込を防ぐように適切なアトミック実行範囲を設定する必要がある．

「実験の目的」

複数変数が含まれる表明はよくモデルに見られる．解析ツールではそのようなモデルに対しても適用でき，適切な修正方法が提示できることを評価する．

反例・正例出力ツールの実行結果は以下に示す.

```
$ ce rc_mv

反例出力の結果：
【出力数】3
【出力先】/Users/chinteki/spin/env/rc_mv_dir/NG/
正例出力の結果：
【出力数】1
【出力先】/Users/chinteki/spin/env/rc_mv_dir/OK/

出力所要時間：1.210 秒
```

誤り特定ツールの適用結果は以下に示す.

```
$ java CEAnalysis rc_mv

誤り箇所特定の結果：
【誤り箇所 1】
次の 4 ステップがアトミック処理になっていない:
[rc_mv:5 行目～rc_mv:8 行目]
正しい実行例: /Users/chinteki/spin/env/rc_mv_dir/OK/rc_mv1.trail.txt

解析所要時間 0.237 秒
```

誤り特定ツールが提示した修正方法でモデルを下記のように修正する.

```
 1 int x,y = 0;
 2
 3 proctype A(){
 4   if
 5     ::d_step{x == 0 && y == 0 ->
 6               x = x + 10;
 7               y = y + 10;
 8               assert(x == 10 && y == 10)}
 9     fi;
10
11 }
12
13 proctype B(){
14     if
15     ::d_step{x == 0 ->
16         x = x + 10}
17     fi;
18 }
19
20 proctype C(){
21     if
22     ::d_step{y == 0 ->
23         y = y + 10;}
24     fi;
25 }
26
27 init{
28       run A();
29       run B();
30       run C()
31 }
```

修正後のモデルに対して，反例・正例出力ツールの実行結果は以下に示す．

```
$ ce rc_mv_fix

指定したモデルから反例が出力されませんでした．
```

適用実験の結果，解析ツールがモデルの誤り箇所が正しく特定できた．解析ツールは表明に複数変数を含むモデルに適用でき，提示された修正方法は適切であることが分かった．

6.1.3　誤り箇所が複数存在するモデル例への適用

評価に使用したモデル「rc_me」は以下に示す.

```
 1 int x,y = 0;
 2
 3 proctype A(){
 4   if
 5     ::x == 0 && y == 0 ->
 6         x = x + 10;
 7         y = y + 10;
 8         assert(x == 10 && y == 10)
 9     fi;
10
11 }
12
13 proctype B(){
14     if
15     ::d_step{x == 0 ->
16         x = x + 10}
17     fi;
18 }
19
20 proctype C(){
21     if
22     ::y == 0 ->
23         y = y + 10;
24         assert(y == 10)
25     fi;
26 }
27
28 init{
29       run A();
30       run B();
31       run C()
32 }
```

「評価例の説明」

この評価例に 6.1.2 のモデルをベースに改造したモデルである．プロセス A とプロセス C がそれぞれ表明を持ち，プロセス A とプロセス C に本来アトミックにするべき処理をそれぞれ外してあるため，２つの誤り箇所を用意してある．

反例・正例出力ツールの実行結果は以下に示す.

```
$ce rc_me

反例出力の結果：
【出力数】10
【出力先】/Users/chinteki/spin/env/rc_me_dir/NG/
正例出力の結果：
【出力数】8
【出力先】/Users/chinteki/spin/env/rc_me_dir/OK/

出力所要時間：2.325 秒
```

「実験の目的」

誤り箇所が複数存在するモデルに対して, 解析ツールが複数誤り箇所を一気に見つけることを評価する.

誤り特定ツールの適用結果は以下に示す.

```
$ java CEAnalysis rc_me

誤り箇所特定の結果：
【誤り箇所 1】
次の 4 ステップがアトミック処理になっていない:
[rc_me:5 行目～rc_me:8 行目]
正しい実行例: /Users/chinteki/spin/env/rc_me_dir/OK/rc_me7.trail.txt
【誤り箇所 2】
次の 2 ステップがアトミック処理になっていない:
[rc_me:22 行目～rc_me:23 行目]
正しい実行例: /Users/chinteki/spin/env/rc_me_dir/OK/rc_me2.trail.txt

解析所要時間 0.435 秒
```

誤り特定ツールが提示した修正方法でモデルを下記のように修正する.

```
 1 int x,y = 0;
 2
 3 proctype A(){
 4   if
 5     ::d_step{x == 0 && y == 0 ->
 6         x = x + 10;
 7         y = y + 10;
 8         assert(x == 10 && y == 10)}
 9     fi;
10
11 }
12
13 proctype B(){
14     if
15     ::d_step{x == 0 ->
16         x = x + 10}
17     fi;
18 }
19
20 proctype C(){
21     if
22     ::d_step{y == 0 ->
23         y = y + 10;
24               assert(y == 10)}
25     fi;
26 }
27
28 init{
29       run A();
30       run B();
31       run C()
32 }
```

修正後のモデルに対して，反例・正例出力ツールの実行結果は以下に示す.

```
$ ce rc_me_fix

指定したモデルから反例が出力されませんでした.
```

適用実験の結果，誤り箇所が複数存在するモデルに対して，解析ツールが誤り箇所を一気に見つけることが確認できた．

6.1.4　表明にローカル変数が含まれるモデル例への適用

評価に使用したモデル「rc_lv」は以下に示す．

```
 1 int x = 5;
 2
 3 proctype threadA()
 4 {
 5
 6 int y = 0;
 7 int z = 0;
 8
 9     if
10     ::x==5
11         ->
12                 y = x * 2;
13                 z = x + y;
14                 assert(y == 10)
15     fi
16 }
17
18
19 proctype threadB()
20 {
21     x = 4;
22 }
23
24 proctype threadC()
25 {
26     x = 3;
27 }
28
29 init
30 {
31 run threadA();
32 run threadB();
33 run threadC();
34 }
```

「評価例の説明」

この評価例は表明にローカル変数が含まれるモデルである．このモデルでは，プ

ロセス A にローカル変数 y に関する表明がある．x の値が初期値の 5 のままでプルセス A の処理が開始され，１０行目の条件判定をした後に，他のプロセスに割込まれずに，そのまま１２行目まで実行し続けると，１２行以降 y の値を変更する処理がいない．そのため，本モデルの場合，１０行目から１２行目までをアトミック処理にすれば，race conditions 問題が発生しない．従って，適切な修正方法は１０行目から１２行目までの処理をアトミック処理にすること．

「実験の目的」

表明にローカル変数が含まれるモデルに対して，解析ツールが提示した誤り箇所は正しく，提示した修正方法が適切であることを評価する．

反例・正例出力ツールの実行結果は以下に示す．

```
$ ce rc_lv

反例出力の結果：
【出力数】6
【出力先】/Users/chinteki/spin/env/rc_lv_dir/NG/
正例出力の結果：
【出力数】5
【出力先】/Users/chinteki/spin/env/rc_lv_dir/OK/

出力所要時間：1.569 秒
```

誤り特定ツールの適用結果は以下に示す．

```
$ java CEAnalysis rc_lv

誤り箇所特定の結果：
【誤り箇所 1】
次の 2 ステップがアトミック処理になっていない:
[rc_lv:10 行目～rc_lv:12 行目]
正しい実行例: /Users/chinteki/spin/env/rc_lv_dir/OK/rc_lv1.trail.txt

解析所要時間 0.243 秒
```

誤り特定ツールが提示した修正方法でモデルを下記のように修正する．

```
 1 int x = 5;
 2
 3 proctype threadA()
 4 {
 5
 6 int y = 0;
 7 int z = 0;
 8
 9     if
10     ::d_step{x==5
11         ->
12                 y = x * 2;}
13                 z = x + y;
14                 assert(y == 10)
15     fi
16 }
17
18
19 proctype threadB()
20 {
21     x = 4;
22 }
23
24 proctype threadC()
25 {
26     x = 3;
27 }
28
29 init
30 {
31 run threadA();
32 run threadB();
33 run threadC();
34 }
```

修正後のモデルに対して，反例・正例出力ツールの実行結果は以下に示す.

```
$ ce rc_lv_fix

指定したモデルから反例が出力されませんでした.
```

適用実験の結果, 解析ツールが予想通りの特定結果と修正方法を提示することが

できた. 解析ツールは表明にローカル変数が含まれたモデルにも適用できるが分かった.

6.1.5 表明がモニターで定義されるモデル例への適用

評価に使用したモデル「rc_am」は以下に示す.

```
 1 int x = 0;
 2
 3 proctype A(){
 4   if
 5     ::x == 0 ->
 6               x = x + 10;
 7               assert(x == 10)
 8     fi;
 9 }
10
11 proctype B(){
12     if
13     ::x == 0 ->
14               x = x + 10;
15               assert(x == 10);
16     fi;
17 }
18
19 proctype monitor(){
20     assert(x == 0 || x == 10);
21 }
22
23 init{
24       run A();
25       run B();
26       run monitor();
27 }
28
```

「評価例の説明」

この評価例ではどの場合でもグレバル変数 x の値が 0 か 10 のどちらかにある性質を記述する表明が独立したプロセス monitor() （名前に特に意味はない）に置かれた. この追加されたプロセスは, processA と processB とともに走る. このプロセスは 1 ステップを実行し終了するが, これはシステムの実行中のどのタイミングでも実行する可能性がある.Promela によりモデル化されたシステム,および,

Spin により検証されるシステムは，完全な非同期である．よって，*Spin* による検証は 3 つのプロセスの可能なすべてのタイミングを検証の対象とするため，この assert は他の 2 つのプロセスの生存期間のあらゆるタイミングで評価される．もし検証プログラムが違反を報告しなかった場合は，われわれは assert 違反を犯すような実行シーケンスが（3 つのプロセスの実行スピードによらず）まったくないことと結論づける事ができる．monitor プロセスによるシステムの不変条件の妥当性をチェックする方法は洗練された方法である．

「実験の目的」

システム実行中どのタイミングでもある性質を持つことを確認する時に時相論理式を書くことが多いが，評価例のように表明をモニタープロセスとして記述することもしばしば見られる．本実験は表明がモニターで定義されたモデルに対しても解析ツールが適用できることを評価する．

反例・正例出力ツールの実行結果は以下に示す．

```
$ ce rc_am

反例出力の結果：
【出力数】12
【出力先】/Users/chinteki/spin/env/rc_am_dir/NG/
正例出力の結果：
【出力数】20
【出力先】/Users/chinteki/spin/env/rc_am_dir/OK/

出力所要時間：1.841 秒
```

誤り特定ツールの適用結果は以下に示す．

```
$ java CEAnalysis rc_am

誤り箇所特定の結果：
【誤り箇所 1】
次の 3 ステップがアトミック処理になっていない:
[rc_am:5 行目～rc_am:7 行目]
正しい実行例: /Users/chinteki/spin/env/rc_am_dir/OK/rc_am4.trail.txt
【誤り箇所 2】
次の 3 ステップがアトミック処理になっていない:
[rc_am:13 行目～rc_am:15 行目]
正しい実行例: /Users/chinteki/spin/env/rc_am_dir/OK/rc_am16.trail.txt

解析所要時間 0.490 秒
```

誤り特定ツールが提示した修正方法でモデルを下記のように修正する.

```
 1 int x = 0;
 2
 3 proctype A(){
 4   if
 5     ::d_step{x == 0 ->
 6               x = x + 10;
 7               assert(x == 10)}
 8     fi;
 9 }
10
11 proctype B(){
12     if
13     ::d_step{x == 0 ->
14               x = x + 10;
15               assert(x == 10);}
16     fi;
17 }
18
19 proctype monitor(){
20     assert(x == 0 || x == 10);
21 }
22
23 init{
24       run A();
25       run B();
26       run monitor();
27 }
28
```

修正後のモデルに対して，反例・正例出力ツールの実行結果は以下に示す.

```
$ ce rc_am_fix

指定したモデルから反例が出力されませんでした.
```

適用実験の結果, 解析ツールは表明がモニターで定義されるモデルにも適用できるが分かった.

6.1.6　プロセス数多いモデル例への適用

評価に使用したモデル「rc_lp」は以下に示す.

```
 1 int x = 5;
 2
 3 proctype threadA()
 4 {
 5     int y = 0;
 6     if
 7     ::x==5
 8         ->
 9               y = x * 2;
10               x =x + 1;
11               assert(y == 10)
12     fi
13 }
14
15 proctype threadB()
16 {
17     int y = 0;
18     x = 4;
19     y = x;
20 }
21
22 proctype threadC()
23 {
24     x = 4;
25 }
26
27 init
28 {
29 run threadA();
30 run threadB();
31 run threadC();
32 run threadC();
33 run threadC();
34 run threadC();
35 run threadC();
36 run threadC();
37 run threadC();
38 run threadC();
39 }
```

「評価例の説明」

この評価例は 6.14 節の評価モデル「rc_lv」をアレンジして，起動プロセス数を増やしたモデル例である．モデルでは１０個のプロセスが起動されている．各プロセスの処理が切り替わって実行されるため，割込が発生する確率が高くなり，数多くの反例が出力されると予想される．

「評価の目的」

解析ツールは，起動プロセス数が多いモデルに対しても適用できることを評価すること．また，数多くの反例を解析ツールで解析するには所要時間が適切であること．

反例・正例出力ツールの実行結果は以下に示す．

```
$ ce rc_lp

反例出力の結果：
【出力数】767
【出力先】/Users/chinteki/spin/env/rc_lp_dir/NG/
正例出力の結果：
【出力数】768
【出力先】/Users/chinteki/spin/env/rc_lp_dir/OK/

出力所要時間：35.000 秒
```

誤り特定ツールの適用結果は以下に示す．

```
$ java CEAnalysis rc_lp

誤り箇所特定の結果：
【誤り箇所 1】
次の 2 ステップがアトミック処理になっていない:
[rc_lp:7 行目～rc_lp:9 行目]
正しい実行例: /Users/chinteki/spin/env/rc_lp_dir/OK/rc_lp1.trail.txt

解析所要時間 6.226 秒
```

誤り特定ツールが提示した修正方法でモデルを下記のように修正する.

```
 1 int x = 5;
 2
 3 proctype threadA()
 4 {
 5     int y = 0;
 6     if
 7     ::d_step{ x==5
 8         ->
 9                 y = x * 2;}
10                 x =x + 1;
11                 assert(y == 10)
12     fi
13 }
14
15 proctype threadB()
16 {
17     int y = 0;
18     x = 4;
19     y = x;
20 }
21
22 proctype threadC()
23 {
24     x = 4;
25 }
26
27 init
28 {
29 run threadA();
30 run threadB();
31 run threadC();
32 run threadC();
33 run threadC();
34 run threadC();
35 run threadC();
36 run threadC();
37 run threadC();
38 run threadC();
39 }
```

修正後のモデルに対して，反例・正例出力ツールの実行結果は以下に示す．

```
$ ce rc_lp_fix

指定したモデルから反例が出力されませんでした．
```

適用実験の結果，解析ツールがモデルの誤り箇所が正しく特定できた．解析ツールはプロセス数が１０個程度のモデルに適用できることが分かった．また，解析に１分間未満で終了できたため，当モデル例と同じレベルのモデルに適用する時に，性能的に問題がないことが確認できた．

6.2 典型的なモデル例への適用

本節では，典型的なモデル例に対して解析ツールの適用結果を述べる．

6.2.1 「並行システムの変数更新問題」モデルへの適用

評価に使用したモデル「incrementer」は以下に示す．

```
 1  #define NUMPROCS 2
 2
 3  byte counter = 0;
 4  byte progress[NUMPROCS];
 5
 6  proctype incrementer(byte me)
 7  {
 8      int temp;
 9
10      temp = counter;
11      counter = temp + 1;
12      progress[me] = 1;
13  }
14
15  init {
16      int i = 0;
17      int sum = 0;
18
19      atomic {
20          i = 0;
21          do
22          :: i < NUMPROCS ->
23              progress[i] = 0;
```

```
24                  run incrementer(i);
25              i++
26          :: i >= NUMPROCS -> break
27          od;
28      }
29      atomic {
30          i = 0;
31          sum = 0;
32          do
33          :: i < NUMPROCS ->
34              sum = sum + progress[i];
35              i++
36          :: i >= NUMPROCS -> break
37          od;
38          assert(sum < NUMPROCS || counter == NUMPROCS)
39      }
40  }
```

「評価例の説明」

この評価例にインターネット上公開された並行プログレミング教科書 [8] に載ってあるモデル例. モデルには並行システムによくある複数プロセスが同じ共有変数に対するカウントアップ処理が記述されている．複数プロセスが全て立ち上げ，かつカウントアップ処理が実行された後に共有変数の値がプロセスの数になる性質が表明で書かれている.

「実験の目的」

本実験では，自作モデルではなく，教科書に載ってある race conditions 問題のモデルをそのまま解析ツールの入力にし，適用できるかを評価する.

反例・正例出力ツールの実行結果は以下に示す.

```
$ ce incrementer

反例出力の結果：
【出力数】1
【出力先】/Users/chinteki/spin/env/incrementer_dir/NG/
正例出力の結果：
【出力数】26
【出力先】/Users/chinteki/spin/env/incrementer_dir/OK/

出力所要時間：1.790 秒
```

誤り特定ツールの適用結果は以下に示す.

```
$ java CEAnalysis incrementer

誤り箇所特定の結果：
【誤り箇所 1】
次の 2 ステップがアトミック処理になっていない:
[incrementer:10 行目～incrementer:11 行目]
正しい実行例: /Users/chinteki/spin/env/incrementer_dir/OK/incrementer10.trail.txt

解析所要時間 0.722 秒
```

誤り特定ツールが提示した修正方法でモデルを下記のように修正する.

```
1   #define NUMPROCS 2
2
3   byte counter = 0;
4   byte progress[NUMPROCS];
5
6   proctype incrementer(byte me)
7   {
8       int temp;
9
10      d_step{temp = counter;
11      counter = temp + 1;}
12      progress[me] = 1;
13  }
14
15  init {
16      int i = 0;
17      int sum = 0;
18
19      atomic {
20          i = 0;
21          do
22          :: i < NUMPROCS ->
23              progress[i] = 0;
24                  run incrementer(i);
25                  i++
```

```
26          :: i >= NUMPROCS -> break
27          od;
28      }
29      atomic {
30          i = 0;
31          sum = 0;
32          do
33          :: i < NUMPROCS ->
34              sum = sum + progress[i];
35              i++
36          :: i >= NUMPROCS -> break
37          od;
38          assert(sum < NUMPROCS || counter == NUMPROCS)
39      }
```

修正後のモデルに対して，反例・正例出力ツールの実行結果は以下に示す.

```
$ ce incrementer_fix

指定したモデルから反例が出力されませんでした.
```

適用実験の結果，解析ツールが race conditions 問題における典型的な「並行システムの変数更新問題」モデルにある誤り箇所が正しく特定できることが分かった.

6.2.2 「読み手書き手問題」モデルへの適用

評価に使用したモデル「rw」は以下に示す.

```
 1 #define semaphore byte
 2
 3 semaphore db= 1, mutex =1;
 4
 5 int rc = 0;
 6
 7 active[2] proctype reader(){
 8 again:
 9     if
10     ::rc==0 && db > 0 ->
11         db --;
12         rc = rc+1;
13     ::d_step{rc>0 ->
14         rc = rc+1;}
15     fi;
16
17 //    "read_data_base";
18
19     if
20     ::rc==1 ->
21         db++;
22         rc = rc -1;
23     ::rc>1 ->
24         rc = rc -1;
25     fi;
26     goto again
27 }
28
29 active[1] proctype writer(){
30 again:
31     skip;
32     d_step{
33     db > 0-> db --;
34
35 // "write_data_base";
36
37     assert(rc==0);
38     db++;
39     }
40     goto again
41 }
```

「評価例の説明」

この評価例は並行システムによくあるリーダライタ問題のモデルである．
システムの振る舞いに以下な特徴を持つことを予想されている．

- ２つの読み手と１つの書き手がいって，DB に対するアクセスをする
- ２つの読み手が同時に DB への読み込み可能
- 書き手の書き込みをしている間に読み手の読み込処理は不可

提案手法の有効性を評価するため，読み手プロセスがデータベースへの読込みをする前後の処理に，本来アトミックにするべき４箇所のうち，１箇所（13 行目～14 行目）をアトミックにして，残りの３箇所（10 行目～11 行目，20 行目～22 行目，23 行目～24 行目）を意図的に割込まれるようにさせている．

「実験の目的」

本実験に使用しているモデルの振る舞いは比較的に複雑である．また，読み手と書き手の処理は循環に行われる．本実験は解析ツールがこのようなモデルに対しても適用できることを評価する．また，既にアトミックにしている部分について，誤って修正方法として提示しないことも確認する．

反例・正例出力ツールの実行結果は以下に示す．

```
$ ce rw 40

反例出力の結果：
【出力数】2633
【出力先】/Users/chinteki/spin/env/rw_dir/NG/
正例出力の結果：
【出力数】732
【出力先】/Users/chinteki/spin/env/rw_dir/OK/

出力所要時間：84.499 秒
```

評価に使う読み手書き手モデルの処理が循環に行われるため，反例・正例出力ツールで反例・正例を出力させる時に，解析に必要以上の出力を抑止するため，状態空間の探索深さを 40 にして出力させた．

誤り特定ツールの適用結果は以下に示す.

```
$ java CEAnalysis rw

誤り箇所特定の結果：
【誤り箇所 1】
次の 3 ステップがアトミック処理になっていない:
[rw:20 行目～rw:22 行目]
正しい実行例: /Users/chinteki/spin/env/rw_dir/OK/rw706.trail.txt
【誤り箇所 2】
次の 2 ステップがアトミック処理になっていない:
[rw:23 行目～rw:24 行目]
正しい実行例: /Users/chinteki/spin/env/rw_dir/OK/rw715.trail.txt
【誤り箇所 3】
次の 2 ステップがアトミック処理になっていない:
[rw:10 行目～rw:11 行目]
正しい実行例: /Users/chinteki/spin/env/rw_dir/OK/rw98.trail.txt

解析所要時間 18.297 秒
```

誤り特定ツールが提示した修正方法でモデルを下記のように修正する.

```
 1 #define semaphore byte
 2
 3 semaphore db= 1, mutex =1;
 4
 5 int rc = 0;
 6
 7 active[2] proctype reader(){
 8 again:
 9     if
10     ::d_step{rc==0 && db > 0 ->
11         db --;}
12                 rc = rc+1;
13     ::d_step{rc>0 ->
14         rc = rc+1;}
15     fi;
16
17 //    "read_data_base";
18
19     if
```

```
20     ::d_step{rc==1 ->
21       db++;
22            rc =   rc -1;}
23     ::d_step{rc>1 ->
24       rc = rc -1;}
25     fi;
26     goto again
27 }
28
29 active[1] proctype writer(){
30 again:
31     skip;
32     d_step{
33     db > 0-> db --;
34
35 // "write_data_base";
36
37     assert(rc==0);
38     db++;
39     }
40     goto again
41 }
```

修正後のモデルに対して，反例・正例出力ツールの実行結果は以下に示す．

```
$ ce rw_fix

指定したモデルから反例が出力されませんでした．
```

適用実験の結果，解析ツールが race conditions 問題における典型的な「読み手書き手問題」モデルにある誤り箇所が正しく特定でき，ツールが提示した修正方法も予想と一致する結果になっている．

6.3 評価・考察

今まで，モデル検査器が出力される反例を解析し，race conditions 問題におけるモデルの誤り箇所を自動的に特定する手法を提案した．また，提案手法に基づいて実装した解析ツールを使って，評価用のモデルと典型的なモデルに適用実験してみ

た．

実験の結果，提案手法及び解析ツールにおけるは以下の利点があることが分かった．

- 提案手法は有効である．実験に使用したモデルに埋め込んだ race conditions 問題の全誤り箇所が解析ツールに自動に特定され，解析ツールが提示した修正方法に従ってモデルを修正して確認した結果，モデル検査器から反例が出力されなくなることが確認でき，提案手法はモデルにある race conditions 問題の誤り箇所を正しく特定できたことが分かった．

- 解析ツールが提示した修正方法は過不足がない，適切である．表明にローカル変数が含まれるモデルなどの適用や，読み手書き手問題モデルの適用などで，必要なアトミックにするべき範囲のみ修正方法として提示された．

- 誤り箇所が複数存在する場合，提案手法の適用によって，複数の誤り箇所を一気に発見できる．

- 提案手法は既存のモデルがそのまま入力とすることができ，反例・正例の出力，誤り箇所の特定，修正方法の提示一連の処理が自動的に行うことが可能となっている．提案手法は一定の実用性を持つ手法である．

- 評価に使用したモデルの反例・正例の出力と誤り特定のいずれも短時間で処理でき，提案手法に割込箇所一覧を先に纏めるなどの対処は効率向上に繋がることが分かった．

評価用モデルの適用実験中で，提案手法及び解析ツールにおける以下の問題を直面した．

- モデルに循環処理が含まれた場合やモデルの状態数が大きいの場合など，膨大な数の反例と正例が出力され，全ての反例と正例の出力は現実ではないことが分かった．その時にモデル検証に使う探索深さを予めチュニングし，適切な探索深さを指定する必要があることが分かった．

- 提案手法ではモデルソースに対して細か構文解析を行わないため，評価用モデルのソースに else 文の条件判定が書かれた場合，反例より解析用情報を正しく抽出できないことが分かった．この問題を解決するには，モデルに else 文を明確的な条件判定を置換した．

- また，提案手法では誤り特定結果を提示する時に，行数を提示するため，条件判定と条件判定後の処理の複数行を改行せず一行に記述された場合も，解析ツールの提示結果は適切ではないことも分かった．そのため，解析ツールを利用する場合，提供対象モデルの実装は条件判定と条件判定後処理をそれぞれの行にして，正確な誤り箇所を特定するために，1 処理ステップを 1 行にする必要がある．

上述のように，本研究は反例よりモデルの誤りを自動的に特定する試練であり，これまでの適用実験結果を考察した結果，モデルを一定の記述ルールに従って記述した上，特定問題に関する具体的な領域知識を加えれば，モデルの誤り自動特定が可能であることが分かった．

第７章　おわりに

7.1　まとめ

今まで，モデル検査器より出力される反例の自動解析について，既存研究では誤りの自動特定までは至っていないが，本研究では race conditions という並行計算に重要な問題に対し，問題の特徴を付加情報とする誤りの自動特定法を提案した．また，提案手法に従って反例解析ツールを実装し，race conditions 問題のモデル例に適用してみた．適用実験の結果，取り上げたモデル例の誤りが自動的に特定でき，提案手法は有効であることが分かった．

7.2　今後の課題

提案手法では解析対象モデルの全反例と全正例を出力されるため，モデルから出力される反例または正例の数が膨大な数である場合，状態区間の探索深さを指定するか，モデルを抽象化して状態数を減らすかなどのチューニングが必要となる．また，本研究では検証する性質を表明に限定しているため，モデル検査に時相論理式などで性質を記述する場合の反例解析は今後の課題とする．

本研究では race conditions という特定な問題に対して，反例に基づいて誤りの自動特定を行った．今後，race conditions 以外にも様々な問題の特徴を記述言語で記述して，記述された問題の特徴に基づいて反例の自動解析研究が期待される．また，それらの研究をベースにして，反例の自動分類や，モデルの自動修正なども期待されると考えている．

謝辞

本論文は筆者が北陸先端科学技術大学院大学 情報科学研究科 博士前期課程に在籍中の研究成果をまとめたものである．

本研究の遂行にあたって終始，ご指導賜りました同大学院 青木利晃准教授に厚くお礼を申し上げます．また，審査委員として有益な助言を頂いた同大学院 二木厚吉教授，鈴木正人准教授に感謝の意を示します．また，研究に関する議論に応じてくださった青木研究室 東京サテライトの鈴木尚志さん，大野乾さん，太田十字光さんに感謝いたします．

最後に，長い間社会人学生の私を支え続けてくれた家族に感謝します．

参考文献

[1] A. Groce and W. Visser : *What went wrong: Explaining counterexamples*, In SPIN Workshop on Model Checking of Software, pages 121-135. Portland, OR (2003)

[2] T. Ball, M. Naik and S. K. Rajamani : *From symptom to cause: localizing errors in counterexample traces*, POPL '03, pages 97-105. (2003)

[3] A. Groce, S. Chaki, D. Kroening and O. Strichman : *Error explanation with distance metrics*, International Journal on Software Tools for Technology, Vol. 8, No. 3, pages 229-247. (2006)

[4] 熊澤努, 玉井哲雄 : *モデルに基づく誤り特定と反例修正候補の提示*, ソフトウェアエンジニアリングシンポジウム. (2009)

[5] E. Clarke and H. Veith : *Counterexamples revisited : Principles, Algorithms, Applications*. In Proc. International Symposium on Verification in Honor of Zohar Manna, volume2772 of LNCS, (2003)

[6] R. H. Netzer, and B. P. Miller : *What are race conditions? some issues and formalizations*. ACM Letters on Programming Languages and Systems, 1(1):74–88, (1992)

[7] G. J. Holzmann. *THE SPIN MODEL CHECKER*. Addison-Wesley, 2nd edition, (2005)

[8] Paul E. McKenney : *Is Parallel Programming Hard, And, If So, What Can You Do About It?*, Linux Technology Center IBM Beaverton, pages 282. (2011)