平成 24 年度

修士学位論文

モデル検査器 Java Pathfinder における実行トレース出力の改善

Improvement of the execution trace output of Java Pathfinder

1155065　　菅 優也

指導教員　　高田 喜朗

2013 年 3 月 1 日

高知工科大学大学院 工学研究科 基盤工学専攻

情報システム工学コース

要 旨

モデル検査器 Java Pathfinder における実行トレース出力の改善

菅 優也

プログラムの検証手法の 1 つに，モデル検査がある．モデル検査とは，形式システムをアルゴリズム的に検証する方法であり，ハードウェアやソフトウェアの設計から導出されたモデルが形式仕様を満たすかどうかについて網羅的に検証を行う．

Java プログラムのモデル検査を行うモデル検査器の 1 つに Java Pathfinder がある．Java Pathfinder は通常の Java 仮想マシン上に独自の仮想マシンを構築し，その上で対象プログラムそのものの網羅探索を行うため，モデル記述言語を用いたモデルの作成が不要である．

Java Pathfinder ではエラー検出の際，経路情報のみが表示され，エラー検出時の状況が把握し辛い．そこで，本研究では状況の把握とエラー原因の特定を容易にするため，実行開始からエラー検出までの各地点の情報を表示する方法を提案した．この実現方法として，探索終了後の仮想マシン自体を操作して情報を取得する方法と，探索時に情報を収集し，エラー検出時に収集した情報から状況を再現する方法の 2 通りの方法を試した．

結果として，後者の方法により実行開始からエラー検出時までのトレース情報の表示を実現することができた．また，被験者に実際に使用してもらった結果，検出したエラー情報の追加や，地点変更時に変更のあった変数情報の強調表示，表示項目自体の説明などが必要との意見があり，表示項目についてさらなる改善が必要であることがわかった．

キーワード　モデル検査, Java Pathfinder, 網羅探索

Abstract

Improvement of the execution trace output of Java Pathfinder

Kan Yuya

Model checking is one of the techniques to formally verify a program. Model checking performs an exhaustive verification in which a model derived from the specification of hardware and software is verified whether it satisfies a formal specification.

Java Pathfinder is one of software model checking tools that performs verification for Java programs. Java Pathfinder has own virtual machine on the Java virtual machine. Java Pathfinder does not require a user to create a model using a model description language because it can perform exhaustive verification of the target program on its own virtual machine.

For users of Java Pathfinder, it is not easy to grasp what happens when an error is detected, because it shows only routing information for the error. To facilitate understanding the situation and identifying the cause of the error, in this study we have proposed a method to display information for each point of the error trace from the initial point to the error point. We tried two implementations. One is to obtain information by operating the virtual machine itself after the end of the search, and the other is to recreate the state information using collected information at the time of error detection. As a result, we were able to achieve the display of trace information from the initial point to the error point by the latter method.

In addition, human subjects gave some comments on the proposed tool. For example, a subject said that the tool have to highlight the variable information which has changed. The comments suggest that there is a need for further improvement on the

displayed items.

key words model checking, Java Pathfinder, exhaustive verification

目次

図目次

表目次

第 1 章

序論

近年多くのシステムで分散処理や並行処理が使用されている．これにより，複雑な処理や膨大な処理を短時間で実行することが可能となった．一方で，分散・並行処理を使用したシステムの検証が問題となっている．分散・並行処理では複数の処理主体が存在し，これらの処理主体が資源を共有することで処理が行われる．しかし，処理主体が資源にアクセスする順番は非決定的であり，特定の順番でアクセスしたときのみエラーとなる場合もある．

分散・並行処理を使用したシステムを正確に検証するためには，各処理主体が資源へアクセスするすべての組み合わせについて検証しなければならない．しかし，処理主体が増加するにつれ組み合わせは階乗的に増加するため，手動による検証手法では困難である．

新しい検証方法として，モデル検査が注目されている．モデル検査とは，形式システムの検証手法の 1 つで，ハードウェアやソフトウェアの設計からモデルを導出し，モデルが形式仕様を満たすかどうかについて網羅的に検証を行う [1]．モデル検査による検証はモデル検査器と呼ばれるソフトウェアを使用して自動的に行われる．ほとんどのモデル検査器では，モデルの記述にハードウェア記述言語や専用の言語を使用するため，専用の知識が必要となる [2]．このモデル検査を使用することで，分散・並行処理を行うシステムを網羅的に検証することが可能となる．

Java プログラムのためのモデル検査器の 1 つに Java Pathfinder(以降 JPF) がある．JPF は，デッドロックやデータ競合など，主にスレッド間で発生するエラーを検証するために使用される．JPF の大きな特徴として，通常の Java 仮想マシン上に Java バイトコードで記述された独自の仮想マシンを実装している (図 1.1，1.2)[3][4]．この独自の仮想マシンでは，プログラムから実行の分岐点の識別や，記録した状態への復元が可能である．これら

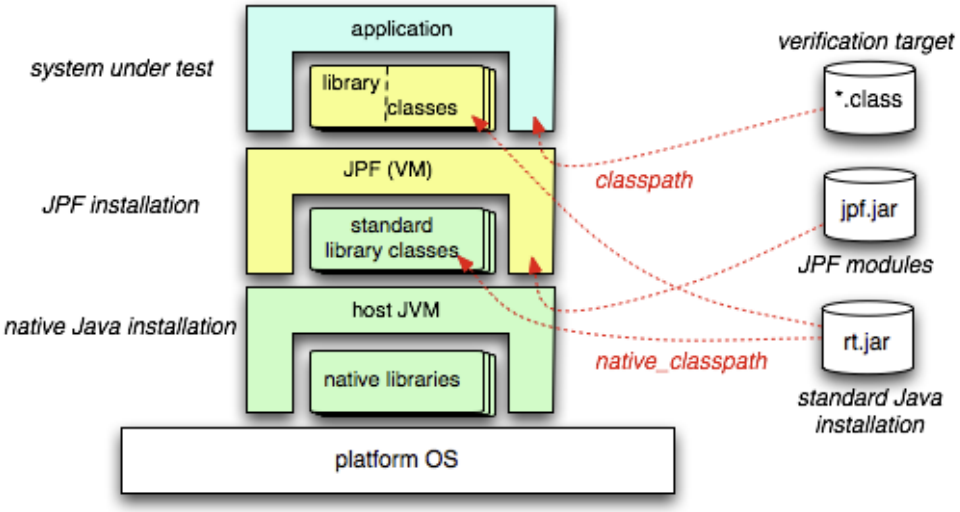

図 1.1　JPF の構造

の機能を使用し，すべての経路について反復実行によって検証するため，モデルを記述する必要は無く，専用の知識は不要である．

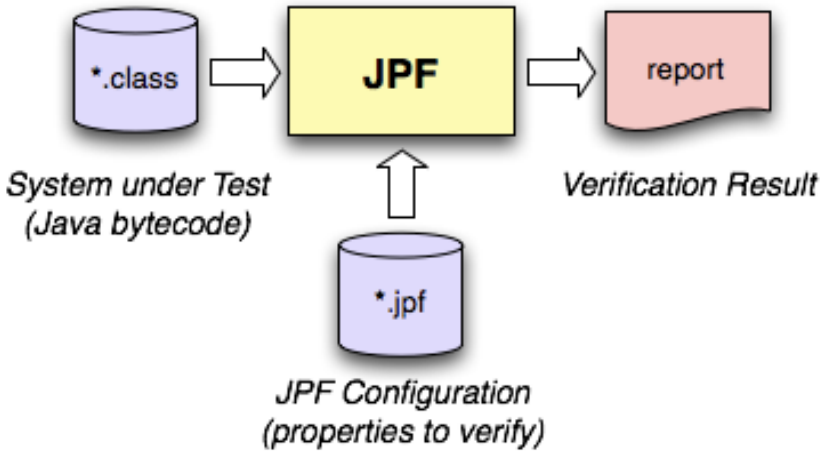

図 1.2　JPF の実行

JPF ではエラーを検出すると，検出時の状況が出力される．また，追加設定により実行開始からエラー検出時までに探索した状態や実行した命令を経路情報として出力することができる（以降トレース出力と呼ぶ）．このトレース出力を基に，プログラム中で不具合が発生している箇所を特定し修正を行う．

JPF の問題点として，トレース出力がわかりづらいという問題がある．このためエラー

検出時の状況を把握し辛く，修正作業に時間がかかってしまう．

JPF の出力を改善しデバッグを容易にするための研究として，[5] がある．この研究では，文字列として表示されるトレース出力ではなく状態遷移モデルに着目し，システム全体の状態遷移グラフとその内部で動作する各スレッドごとの状態遷移グラフの表示を行う．この研究では，システムの有限状態空間モデルをより見やすくし，デバッグの際にエラーの原因を特定しやすい GUI を提供することを目的としている．しかし，状態空間全体を表示することは必ずしもデバッグを容易にするとは限らない．例えば，[5] では各状態をシンプルに表示することで状態空間全体を把握しやすくしているが，各状態がプログラム中のどの部分の処理実行時の状態であるか把握し辛い．一方で，各状態について詳細に表示すると，全体として表示される情報量が増えるため状態空間全体を把握し辛くなる．また，状態遷移グラフ自体に慣れていないという場合や，エラー原因特定のために全状態空間の情報を必要としない場合も考えられる．

本研究では，表示する情報をエラーが検出された経路に限定してトレース出力の改善を行う．改善方法として，既存のトレース出力とともに，経路中の各地点の状況を把握するために必要な情報の提示を行う．この方法により，実行開始からエラー検出までの経路中の状況変化を把握させることで，プログラム中で予期せぬ処理を行っている部分の特定にかかる時間を短縮させることを目的とする．

第 2 章

JPF の問題点と解決策

この章では JPF の問題点であるトレース出力がわかりづらいという問題について具体例を用いて説明する.

ここでは例として，図 2.1 のプログラムの検証を行う．このプログラムでは，複数のスレッドから同じグローバル変数を同時にインクリメントすることで，データ競合の可能性を含ませている.

```
public class SumCheck {
    static int sum = 0;

    static public class ThreadSum extends Thread {
        ThreadSum() {
        }
        public void run() {
            sum++;
        }
    }

    public static void main(String[] args) {
        ThreadSum[] s = new ThreadSum[3];
        try {
            for(int i = 0; i < 3; i++) {
                s[i] = new ThreadSum();
                s[i].start();
            }
            for(int i = 0; i < 3; i++) {
                s[i].join();
            }
        }catch(Exception e) {}
        assert(sum == 3);
    }
}
```

図 2.1　サンプルプログラム

図 2.1 の検証時に出力される情報として，デフォルトで出力される情報と，追加設定によるトレース出力について説明する．

2.1　デフォルトの出力

デフォルトで出力される情報について説明する．このとき出力される情報は図 2.2 のようになる．

```
====================================================== error #1
gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty
java.lang.AssertionError
        at SumCheck.main(SumCheck.java:23)

====================================================== snapshot #1
thread java.lang.Thread:{id:0,name:main,status:RUNNING,priority:5,lockCou
nt:0,suspendCount:0}
  call stack:
        at SumCheck.main(SumCheck.java:23)

====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty "java.lang.Assert
ionError  at SumCheck.main(SumChec..."
====================================================== statistics
elapsed time:       00:00:00
states:             new=29, visited=9, backtracked=22, end=3
search:             maxDepth=16, constraints hit=0
choice generators:  thread=28 (signal=0, lock=1, shared ref=6), data=0
heap:               new=371, released=74, max live=351, gc-cycles=31
instructions:       3781
max memory:         81MB
loaded code:        classes=83, methods=1150
====================================================== search finished: 1
3/01/31 12:46
============================================ search finished:
```

図 2.2　実行結果：デフォルト表示

図 2.2 は，*error #1* でエラーの概要，*snapshot #1* でエラー検出時に存在するスレッドの情報，*results* で検証結果，*statistics* で探索自体の情報がそれぞれ表示されている．この情報を見ると，プログラム中のどの行でエラーを検出したのか知ることができるが，どの部分で予期しない処理が行われたのかを知ることはできない．

2.2　トレース出力

トレース出力について説明する．このとき表示されるトレース出力は長くなるため，エラー検出時から 5 つ前までの状態を図 2.3 に示す．なお，トレース出力全体は付録 A に記載する．

```
------------------------------------------------------ transition #11 thr
ead: 3
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"sharedField" ,2/2,isCasc
aded:false}
  SumCheck.java:8                : sum++;
------------------------------------------------------ transition #12 thr
ead: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"sharedField" ,1/2,isCasc
aded:false}
  SumCheck.java:8                : sum++;
  SumCheck.java:9                : }
       [1 insn w/o sources]
------------------------------------------------------ transition #13 thr
ead: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"terminate" ,1/2,isCascad
ed:false}
       [2 insn w/o sources]
  SumCheck.java:19               : for(int i = 0; i < 3; i++) {
  SumCheck.java:20               : s[i].join();
       [1 insn w/o sources]
------------------------------------------------------ transition #14 thr
ead: 3
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"wait" ,1/1,isCascaded:fa
lse}
  SumCheck.java:8                : sum++;
  SumCheck.java:9                : }
       [1 insn w/o sources]
------------------------------------------------------ transition #15 thr
ead: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"terminate" ,1/1,isCascad
ed:false}
       [2 insn w/o sources]
  SumCheck.java:19               : for(int i = 0; i < 3; i++) {
  SumCheck.java:22               : }catch(Exception e) {}
  SumCheck.java:23               : assert(sum == 3);
       [21 insn w/o sources]
```

図 2.3　実行結果：トレース出力（一部）

図 2.3 のように，その地点で実行したスレッド名やソースコードファイル名，実行中の行番号やソースコード命令などが表示される．しかし，この情報では探索時の状況を把握し辛

い．この理由について以下で説明する．

2.2.1　変数情報の不足

図 2.3 には，変数の値に関する情報が不足している．これにより以下の場合に問題となる．

- 演算処理が正しく行われているか調べたい場合

  探索中に実行した演算処理について，想定した処理が行われているか確認するには変数の値に関する情報が必須である．しかし，変数の値に関する情報が不足しているため，演算前の値や演算後の値，演算に使用した変数の値などの情報の把握が困難である．トレース出力のはじめから変数の値を手動で計算することにより把握することができるが，時間がかかる上１つでも計算ミスがあると誤った情報として認識してしまう．

- 初期化処理に誤りがある場合

  JPF のトレース出力は実行したソースコード情報や遷移した状態情報が中心である．そのため，プログラムによっては変数の初期値が誤っている場合でもトレース出力がほとんど変化しない場合がある．例えば，図 2.1 のプログラムでは，グローバル変数 *sum* の初期値が -1 の場合，必ずアサーションエラーが発生する．しかし，この場合のトレース出力は初期化処理の部分が異なるだけで他の部分は全く同じであり，誤りに気づきにくい．そのため，修正しているつもりで正しく処理を行っている部分に誤った変更を加える可能性がある．

- 同名の変数を使用する場合

  Java では型が違う場合同名の変数を使用することができる．同名の変数はプログラム作成時に混同する可能性があり，あまり使用されない．しかし，この場合ではトレース出力を見てもどちらの変数を用いているのかわからない．

2.2.2　他のスレッド情報の不足

図 2.3 を見ると，実行中のスレッド名は表示されているが実行待機中のスレッド情報がない．このことは，エラーとなる原因を探る際に問題となる．

例えば，図 2.1 のプログラムは，グローバル変数のインクリメントを行うすべてのスレッドの終了を待った後にアサーション処理を行っている．しかし，全スレッドの終了を待機する部分の処理を誤っていた場合，未実行のスレッドが残っている状態でアサーション処理を行い，エラーを検出する可能性がある．

別の例として，エラーとなる原因を予想し修正を行う場合を考える．予想が正しいかチェックするためには，トレース出力から目的の処理を行っている部分を参照する必要がある．図 2.1 のプログラムは単純な処理のため簡単に参照することができるが，より多くのスレッドを使用する，あるいは各スレッドの処理が複雑なプログラムの場合，目的の処理を行う部分を特定するのに時間がかかる．図 2.1 のように，すべてのスレッド終了時の情報を参照する場合は容易であるが，特定のスレッド終了時の場合，スレッドの実行状態についてトレース出力のはじめから順にチェックを行わなければならない．

2.2.3　表示がわかりにくい

トレース出力には実行したソースコード命令が表示される．しかし，出力されるのは実行したソースコード命令 1 行のみであり，処理の前後関係を把握し辛い．例として，図 2.1 のトレース出力のうち，最後に遷移した状態中の表示を扱う（図 2.4）．

図 2.4 での処理は，19 行目の繰り返し命令の終了条件をチェックし，条件を満たしたため次の 22 行目の処理を行っている．しかし，トレース出力のソースコード命令のみを見ると for 文の終了部分が 22 行目の最初の中括弧のように見え，ソースコードの処理を把握し辛い．

別の例として，複雑な処理を行う場合を考える．複雑な処理を行う場合，多くのソースコード命令が必要となる．しかし，JPF による網羅探索ではこの処理の実行中に別の処理

```
------------------------------------------------------ transition #15 thr
ead: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"terminate" ,1/1,isCascad
ed:false}
      [2 insn w/o sources]
  SumCheck.java:19               : for(int i = 0; i < 3; i++) {
  SumCheck.java:22               : }catch(Exception e) {}
  SumCheck.java:23               : assert(sum == 3);
      [21 insn w/o sources]
```

図 2.4　トレース出力（最後に遷移した状態）

が割り込む場合も検証する．この場合，全体の処理はトレース出力中に分散して表示されることになる．そのため，トレース出力を見ても表示されている処理が何を行うものであるかわかりづらい．

これらの問題は，表示される行番号を基にソースコードを参照することで解決可能である．しかし，トレース出力中の各地点ごとにソースコードを参照しなければならず，手間と時間がかかる．

2.3　解決策

上記の問題点について，解決策を説明する．

2.3.1　変数情報の不足

この問題は，不足している変数情報を追加表示することで解決する．ここで必要となるのは，トレース出力中の各地点実行中に存在する変数名と値の情報である．また，同名の変数が存在することも考慮すると型情報も必要となる．

分散・並行処理を行うシステムでは，同期命令により変数がロックされる場合がある．場合によってはデッドロックとなることもあり，正確な状況把握には変数がどのスレッドによってロックされているかという情報も必要である．

これらの情報を，グローバル変数・フィールド変数・ローカル変数について表示することで解決されると考えられる．

2.3.2　他のスレッド情報の不足

この問題は，トレース出力中の各地点実行中に存在するスレッド情報を表示することで解決される．スレッド情報としては，スレッド名，状態，実行済みの行番号，変数情報が挙げられる．ここで扱う状態とは，スレッドの状態のことであり，生成され未実行である，実行中である，他のスレッドによって実行をブロックされているといった情報である．また，スレッドごとの情報としてフィールド変数とローカル変数が必要である．

これらの情報を，トレース出力中の各地点ごとに表示・閲覧させることで解決されると考えられる．

2.3.3　表示がわかりにくい

解決策として，トレース出力で表示されるソースコード命令にインデントを追加する方法が考えられる．ソースコードからインデントを含めた情報を取得・表示することは可能であるが，ソースコードにインデントを加えるかどうかは作成者に依存する．また，この方法ではトレース出力を多少わかりやすくする程度であり，前後の処理を把握することにはつながらない．

この問題については，統合開発環境 Eclipse で Java プログラムをデバッグする際の表示方法を参考にした [6]．Eclipse のデバッグツールでは全体のソースコードから実行中の行を強調表示している．これによって実行中の行とソースコード全体の処理を同時に把握することが可能である．このように，全体のソースコードから実行中の行を強調表示することでこの問題は解決されると考えられる．

第 3 章

提案手法

前章の解決策の実装を行う．以下では情報をどのように表示させるかについて説明する．

JPF には，様々な拡張機能が存在する．その中でも Listener と呼ばれる拡張機能を利用し，情報の提示を行う．Listener とは，探索中および探索終了後に処理を追加する機能であり，いくつかの機能は予め用意されている．各 Listener の処理はソースコードとして記述されており，ユーザ自身で作成することも可能である．これを利用し，探索開始からエラー検出までの経路中の各地点について，情報を表示する Listener を作成する．

作成する Listener で表示する画面は図 3.1 のようになる．また図 3.1 について，左側を拡大したものが図 3.2，右側を拡大したものが図 3.3 となる．

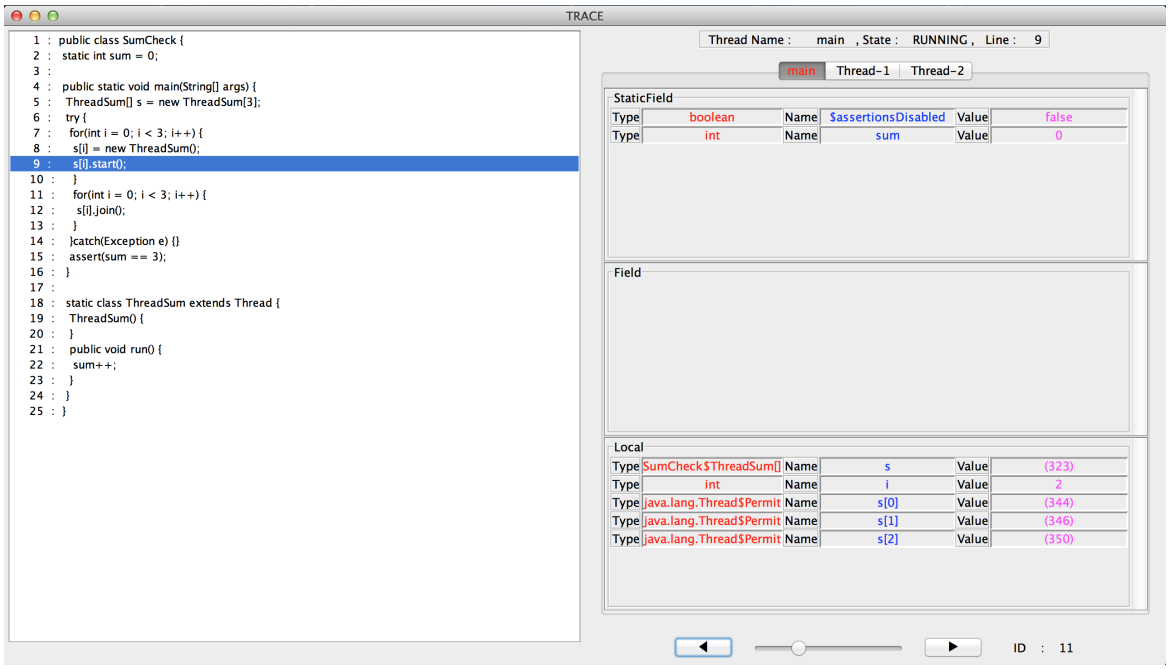

図 3.1　出力画面全体

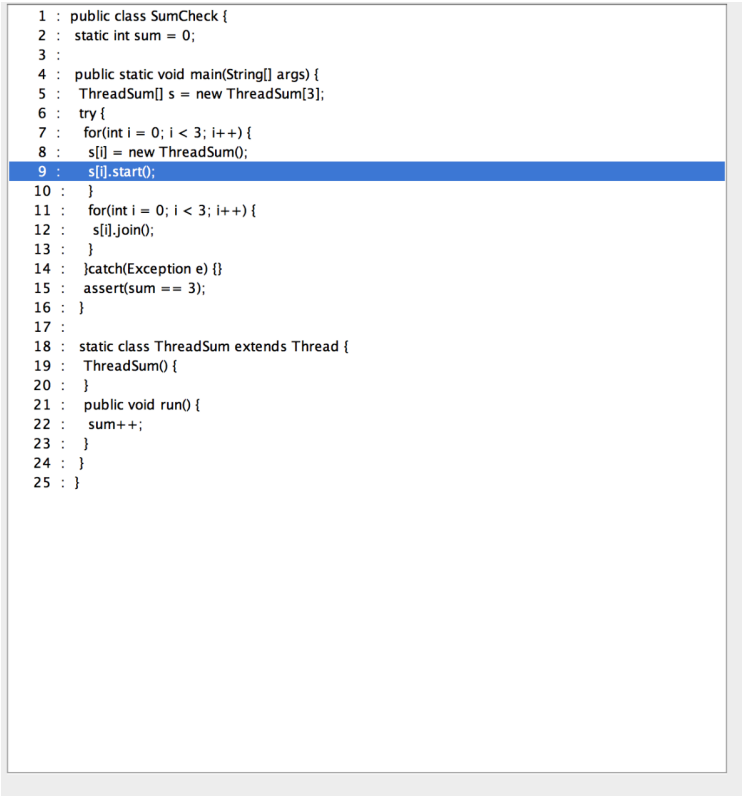

図 3.2　出力画面左側

出力画面の説明は次の通りである．図 3.3 はスレッド情報を，図 3.2 は図 3.3 で表示中のスレッドがソースコードのどの位置まで実行しているのか表示する．また，図 3.3 のタブを切り替えることで，同地点での他のスレッドの情報を表示させる．このときの切り替えに応じて図 3.2 で表示させる内容も変更することで，同地点での各スレッドがソースコード上のどの位置まで実行しているのか一目で分かるようにする．

図 3.3 の 1 つのスレッド情報には，グローバル変数・フィールド変数・ローカル変数を表示する．同地点での他のスレッド情報を知りたい場合，タブを切り替えることにより対象のスレッド情報を表示させる．

実行開始からエラー検出までの経路中のどの地点の情報を表示するかは，図 3.3 下部のボタンおよびスライダーで指定する．ボタンで 1 つ前または 1 つ後の地点へ，スライダーで大きく地点を変化させる．また，スライダーは表示する地点を変更する以外に，現在表示している地点が全体の中のどの地点であるかを把握することに役立つ．

以降では実装方法について説明する．

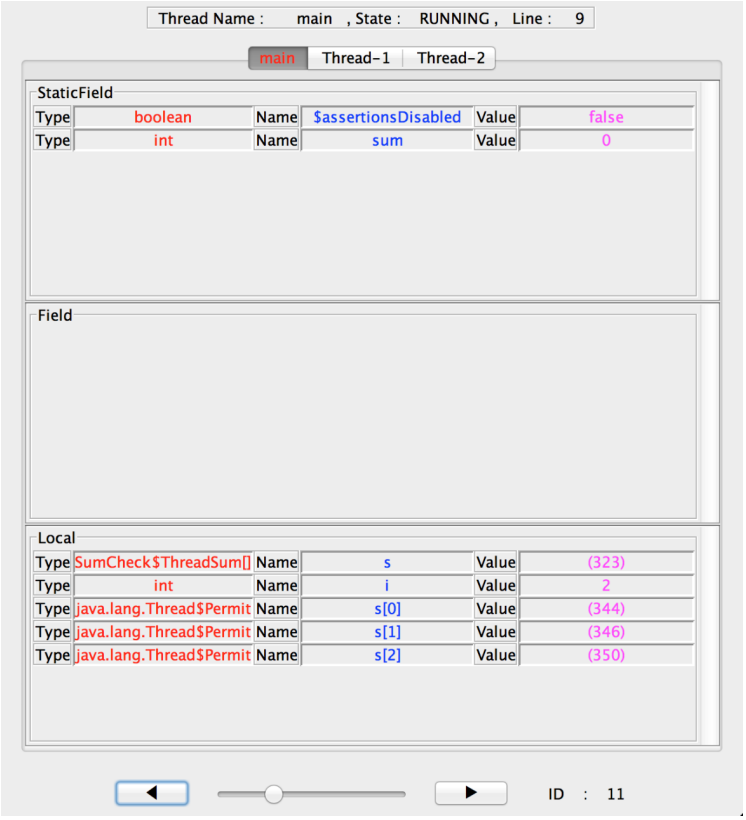

図 3.3　出力画面右側

3.1　JPF の状態

具体的な実装方法を説明するためには JPF で扱う状態についての説明が必要となるため，まず JPF の状態について説明する．

JPF では，1 つの状態をスレッドの実行位置や変数の値，ヒープによって定義しており，これらの情報を基に探索済みの状態であるか判断しながら探索を行う．ここで未探索の状態であると判断すると探索を続けるが，探索済みであると判断した場合，別経路の探索を行う [7][8]．探索中に遷移した各状態には 0 から順に ID が割り当てられ，探索済みの状態には同じ ID が割り当てられる（図 3.4）．

JPF で扱う状態は，単に条件分岐によってわけているのではなく，実行する順番によって全体の処理に影響を与える可能性のある命令によってわけられている．これらの命令は予め定義されており，グローバル変数へのアクセスや同期命令などが該当する．これらの命令が

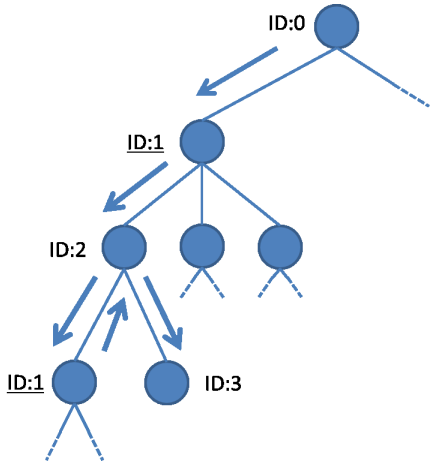

図 3.4　状態空間の探索例

読み込まれると，以降の処理は別の状態として扱う．このため，繰り返しや条件分岐，ローカル変数へのアクセスなど，実行する順番で全体の処理に影響を与えない命令は同じ状態内で処理されることになる．

次に，JPF がプログラム中からどのように状態空間を生成するかについて説明する．3 つのスレッドを持つプログラムにおいて，全スレッドの処理中に全体の処理に影響を与える可能性のある命令が存在しない場合，全体で 6 通りの経路が存在する．また，3 つのスレッドのうちの 1 つだけ，スレッド内の処理に全体の処理に影響を与える可能性のある命令が 1 つ存在する場合，次の 12 通りの経路となる．なお，影響を与える命令が出現するスレッドを 1 とし，出現前を 1-1，出現後を 1-2 として状態を表現する．

```
[1-1, 1-2, 2, 3], [1-1, 2, 1-2, 3], [1-1, 2, 3, 1-2],
[1-1, 1-2, 3, 2], [1-1, 3, 1-2, 2], [1-1, 3, 2, 1-2],
[2, 1-1, 1-2, 3], [2, 1-1, 3, 1-2], [2, 3, 1-1, 1-2],
[3, 1-1, 1-2, 2], [3, 1-1, 2, 1-2], [3, 2, 1-1, 1-2]
```

ここではスレッドが 3 つとなった後の経路として説明したが，実際にはスレッドを 3 つ生成する前の部分で状態分けが行われるため，より多くの経路が存在する．JPF では探索中に状態分けを行うことで全経路の網羅探索を行っている．

JPF では，各状態を表現するために独自の仮想マシンを用いる．この仮想マシン上で，スレッドの実行位置や変数の値などにより各状態を再現することで探索を進める．

3.2　実装方法

ここからは提案手法の実現のために考案した実装方法について説明する．考案した方法は以下の 2 通りである．

- エラー検出後に情報を取得する方法
- 探索中に情報を取得する方法

以降では，これらの具体的な内容について説明する．

3.2.1　エラー検出後に情報を取得する方法

この方法は，エラー検出後にのみ処理を行う．

前節のように，JPF では仮想マシンを使って各状態を表現している．また，一度探索した状態は JPF 内に記録されており，別の経路を探索する場合は記録した情報から状態の復元を行っている．JPF では，エラーを検出すると仮想マシンはエラーを検出したときの状態で停止するため，エラー検出後でもこれらの情報にアクセスすることができる．

これらを利用し，エラー検出後に仮想マシンから経路中に訪れた状態情報を取り出し，仮想マシンをユーザが望む状態へ直接復元し，必要な情報を取得・表示する．

この方法によるメリットとして，探索後にのみ処理を行うため探索時間に影響を与えないこと，新たに情報を格納する必要がないためメモリ使用量を考慮する必要がないことである．デメリットとしては，仮想マシンの状態を実行開始地点からエラー検出地点まで変更するときに時間がかかる恐れがあることである．

この実装案を実現するためには，特定の状態へ遷移させる方法が必要となる．以降では指定した状態へ遷移させる方法の試行結果について説明する．

同じ状態への遷移方法

同じ状態へ遷移させるために，JPF の拡張機能の 1 つである Searcher の動作を調べることで探索中の処理を調査した．

Searcher は探索方法の制御を行う拡張機能であり，Listener 同様 Searcher にも深さ優先探索や幅優先探索のような機能が予め用意されている．これらの各機能の処理はソースコードとしてそれぞれ記述されており，すべてのソースコードで Search クラスを継承している．この Search クラスには，探索中の状態を記録するために必要な情報や，状態遷移に関する処理が記述されている．そこで，Search クラスからどのような処理によって状態遷移が行われているか調査した．

Search クラスより，JPF 独自の仮想マシンを構成するクラス中の forward メソッドによって新しい状態へ遷移，backtrack メソッドによって親状態へ遷移する処理が行われていた．この forward メソッドの処理を調べると，探索中に存在するスレッド数にあわせてスレッド情報を格納する ThreadInfo オブジェクトが生成され，全体の処理に影響を与える可能性のある命令が読み込まれると ChoiceGenerator によって次に実行するスレッドを決定していた．

上記の処理より, ChoiceGenerator を使用して同じ状態へ遷移させるよう試みた．ChoiceGenerator は状態ごとに存在し，状態遷移情報の管理を行っている．また，エラー検出後，探索中に使用したすべての ChoiceGenerator を取得することができる．これにより，探索中に使用した ChoiceGenerator の中から対応するものを適用することで同じ状態へ遷移させようと試みたが，forward 処理時に情報が書き換えられるため別の状態への遷移となった．そこで，ChoiceGenerator 中の情報を制御することで同じ状態へ遷移させるよう試みた．ChoiceGenerator には，遷移先を管理する *counter* 変数と，探索済みであるか管理する *isDone* 変数がある．forward 処理時にはこれらの変数が書き換えられるが，外部からこの変数へアクセスすることや，メソッドを使用して任意の値に変更することはできない．

別の方法として，既存の forward メソッドを使用せず，forward メソッドの処理を模倣し

て遷移させる方法を試みた．この方法では，予め情報を取得し，取得した情報から遷移先の設定を行う．しかし，実行するスレッドを設定する際にエラーが発生し，実現には至らなかった．この原因を特定することはできなかったが，取得したスレッド情報に誤りがある，もしくは次に実行するスレッドを設定する処理に問題があると考えられる．

3.2.2　探索中に情報を取得する方法

この方法は，探索中に情報を取得し，エラー検出後に取得した情報を基に表示を行う．この方法は以下の手順で処理を行う．

1. 探索中に情報を格納
2. エラー検出時に全体の状況を復元
3. 情報の表示

上記の各項目について以降で説明する．

探索中に情報を格納

この項では探索中に情報を格納する方法について説明する．

JPF では，バイトコード単位で命令を実行しながら探索を行う．そのためソースコード上では 1 行の命令でも，バイトコード命令に置き換えると複数行になる．前章で説明した，全体の処理に影響を与える可能性のある命令も，実行されたバイトコード命令をチェックすることで行われる．

JPF では，状態が遷移した，バイトコード命令が実行されたなどの節目に処理を追加できるように Listener 用のインタフェースが定義されている．その中から以下のものを使用する．

- public void instructionExecuted(JVM vm)

  バイトコード命令実行後に呼び出される．

- public void stateAdvanced(Search *search*)

  forward・backtrack 処理実行後（状態遷移後）に呼び出される．

- public void stateBacktracked(Search *search*)

  backtrack 処理実行後に呼び出される．

- public void publishPropertyViolation(Publisher *publisher*)

  エラーを検出した際に呼び出される．

探索中に処理を加えたい場合，上記のものをオーバーライドし，行いたい処理を記述することで反映される．上記のものを利用すれば，命令実行後・状態遷移後・backtrack 処理実行後・エラー検出時の 4 つの節目に行う処理が記述できる．以降ではこのインタフェースを利用した情報取得方法について説明する．

エラー検出までの経路情報の抽出

JPF の探索時に訪れるのは 1 つの経路だけではなく，複数の経路の網羅探索が行われる．そのため，instructionExecuted によってすべての情報を格納しただけでは不要な経路の情報まで表示することになってしまう．そこで，以下のものを利用して探索開始からエラー検出までの経路中の情報のみを取得する．

- バイトコード単位で情報を格納する変数 *byteInfo*
- 状態単位で *byteInfo* を格納するスタック *stateInfo*
- 経路中に存在する *stateInfo* を格納するリスト *pathInfo*

上記の 3 つを利用し，以下のように情報の取得を行う．

1. instructionExecuted を使用し，バイトコード命令が実行されるたびに以下の処理を行う．

   （a）表示に必要な情報を *byteInfo* に格納する．

   （b）情報を格納した *byteInfo* を *stateInfo* に格納する．

2. stateAdvanced を使用し，状態遷移後 *pathInfo* に *stateInfo* を格納し，*stateInfo* を空にする．
3. stateBacktracked を使用し，backtrack 処理実行後 *pathInfo* に最後に格納した *stateInfo* を 1 つ削除する．

この処理は JPF の探索でエラーが検出されると，仮想マシンはエラー検出時の状態で停止することを利用している．このためエラー検出時には経由した状態中の情報のみが格納されており，エラーが検出されなかった場合はすべての情報が破棄されることになる．

格納する情報の削減

前述の方法により，探索開始からエラー検出までの経路中でバイトコードが実行されるたびに情報を格納できるようになった．しかし，バイトコード命令が実行されるたびに情報を格納するとすぐにメモリを使い果たしてしまう．そもそも必要なのはソースコード命令実行直後の情報であり，実行中の情報は不要である．そこで，これらの不要な情報の削除を行った．

説明のために，JPF で int 型のローカル変数の四則演算を実行する際に処理されるバイトコード命令を以下に記す．

- 加算時に実行されるバイトコード命令
  IINC
- 減算時に実行されるバイトコード命令
  IINC
- 乗算時に実行されるバイトコード命令
  ILOAD
  ICONST
  IMUL
  ISTORE
- 除算時に実行されるバイトコード命令
  ILOAD
  BIPUSH

IDIV
ISTORE

これらのバイトコード命令が実行されるたびに情報を格納したが，情報削減のため最後にバイトコードが実行されたときの情報のみを格納する．これは四則演算で実行される IINC または ISTORE 実行後のことである．

JPF では，実行中のバイトコード命令が所属するソースコード命令の行番号を取得することができる．これを利用し，最後に実行された行番号と現在実行中の行番号を比較することで情報の格納を行う．この処理は以下の手順で行う．なお，最後に実行した行番号は *lastLineNumber* に格納するものとして説明する．

1. instructionExecuted が呼び出されたとき，次の処理を行う．
   （a）現在実行中の行番号と *lastLineNumber* を比較する．
      - 等しければ *stateInfo* から最後に格納した *byteInfo* を削除する．
   （b）*byteInfo* に必要な情報を格納する．
   （c）*stateInfo* に *byteInfo* を格納する．
   （d）*lastLineNumber* に現在の行番号を格納する．
2. stateAdvanced が呼び出されたとき，*lastLineNumber* の値を初期化する．

上記のように情報の格納を行うことで，ソースコード中の同じ行で実行されるバイトコード命令のうち，最後に実行されたときの情報のみ残す．しかし，これは同じ状態内での処理でしか機能しない．これは状態遷移が行われると *stateInfo* を *pathInfo* に格納し，*stateInfo* を空にするためである．この問題を解決するために，各スレッドの情報が最後に格納された *pathInfo* のリスト番号を格納する方法を採用した．これにより，*stateInfo* が空の場合，このリスト番号から情報を取得することで，適切に処理することができる．

上記の手順により，ソースコード命令中でバイトコード命令が最後に実行したときの情報のみ残すことが可能となったが，このままでは実行過程で変化した変数情報を格納することができない．そこで，*byteInfo* とは別に，同じソースコード命令中で変化した変数情報を

格納する．この情報は，新しいソースコード命令が実行されたときに初期化される．また，*byteInfo* にはこの情報を変数の変化情報として格納することで情報を反映させる．

格納する情報

ここからは *byteInfo* に格納する情報について説明する．

byteInfo に格納する情報として，以下のものが挙げられる．

- スレッド自体の情報

  スレッド生成時，スレッド ID・スレッド名・スレッドの処理が記述されたソースコードファイルへのパスを記録する．JPF ではスレッドは ID によって管理されているため，スレッド名とスレッド ID を対応させることで探索中に記録する情報量を軽減させる．また，表示する際ソースコードファイルから情報を取得するため，ソースコードファイルへのパスを取得する．これは検査対象のソースコードファイルとは別にスレッドの処理を記述したソースコードファイルが存在する場合があるためである．
- 変数情報

  以下の情報のうち該当するものを格納する．
  - グローバル変数（型・名前・値など）
  - フィールド変数（型・名前・値など）
  - ローカル変数（型・名前・値など）
- 実行中の行番号
- スレッドの状態

変数の変化情報の格納

変数の変化情報の格納方法について説明する．

変数を変化させるバイトコード命令は複数存在するが，予め定義されている．そのため，

特定のバイトコード命令が実行されたときにのみ変数情報を取得する．具体的なバイトコード命令は以下の通りである．

- グローバル変数
  - PUTSTATIC：グローバル変数の格納
- フィールド変数
  - PUTFIELD： フィールド変数の格納
- ローカル変数
  - 基本型
    - ISTORE：boolean, byte, short, int, char 型の値格納
    - LSTORE：long 型の値格納
    - FSTORE：float 型の値格納
    - DSTORE：double 型の値格納
  - 参照型
    - ASTORE：オブジェクトの格納
    - AASTORE：オブジェクト配列の格納
  - 配列
    - IASTORE：byte, short, int 型の配列の値格納
    - LASTORE：long 型配列の値格納
    - FASTORE：float 型配列の値格納
    - DASTORE：double 型配列の値格納
    - CASTORE：char 型配列の値格納
    - BASTORE：boolean 型配列の値格納
- 変数のロック・アンロック情報
  - monitorenter：変数をロック
    - ロック元のスレッド ID

　　∗ ロック対象の変数名

– monitorexit： 変数をアンロック

　　∗ アンロック対象の変数名

上記のバイトコード命令が実行されたときだけ変数情報の格納を行う．前述のように，変数の変化情報は *byteInfo* とは別に格納するため，すべての変化情報を格納することが可能である．

以降では，具体的な値の取得方法について説明する．

変数情報の取得方法

ここでは実際の格納方法について説明する．

変数情報の取得方法は，JPF に付属の Listener の 1 つ NumericValueChecker を参考にした．JPF にはバイトコード命令に応じて処理が記述できるようにインタフェースが定義されている．このインタフェースは InstructionVisitorAdapter クラスで定義されており，このクラスを継承し，目的のものをオーバーライドすることで処理の記述ができるようになっている．例えば，バイトコード命令 ISTORE が実行されたときに処理を記述したい場合，以下のように記述する．

```
class sample extends InstructionVisitorAdapter {
  @Override
  public void visit(ISTORE insn) {
    ---行いたい処理を記述---
  }
}
```

上記のように，変数情報格納用のクラスを用意し，格納処理はすべてこのクラスで行う．

変数情報の取得方法は，実行したバイトコード命令の種類によって異なる．具体的には，グローバル変数・フィールド変数とローカル変数で異なり，さらに基本型，参照型，配列で格納する情報や格納方法が異なる．格納する情報は以下のようになる．

- 基本型の場合
  - 型
  - 名前
  - 値
- 参照型の場合
  - 型
  - 名前
  - アドレス
- 配列の場合
  - 型
  - アドレス
  - 値

配列で名前でなくアドレスを格納しているのは，配列全体は参照型，各要素へのアクセスは配列として扱い，各要素へのアクセス時に配列名を取得することが困難なためである．このため復元時にアドレスから配列名を取得することで表示の際には配列名を表示させる．

グローバル変数・フィールド変数の情報は基本的に FieldInfo オブジェクトから取得する．ただし，基本型の値は命令情報から，アドレスと配列の情報は ElementInfo オブジェクトから取得した．また，配列の値の取得方法は，配列の型によって異なる．

ローカル変数の情報は，基本的に命令情報から取得できる．ただし，基本型の値はスレッド情報から，アドレスや配列は ElementInfo オブジェクトから取得した．ローカル変数でもグローバル変数・フィールド変数同様配列の値取得の際に型情報を使用する．

グローバル変数・フィールド変数については他のスレッドによってロックされる可能性があるため，ロック・アンロック情報を格納する．この情報は ElementInfo オブジェクトから取得する．

3.3　状態の復元

格納した情報を基に，状態の復元を行う．復元方法としては，探索中に格納した情報を基に各地点の状況の再現を行う．具体的には，探索時に格納した情報を実行された順に並べ替え，その後情報を順に取り出し，取り出した情報から仮想マシン内の状態がどのように変化していったのか再現する．再現するためには ThreadInfo のような 1 つのスレッド情報を表現するオブジェクトを用意し，スレッドの生成や終了にあわせて増減させる．また，変数が変化したという情報が取り出されると，対応するスレッドオブジェクト内の変数の値を変更する．また，探索経路中の各地点での状況を格納する配列を用意し，情報を取り出すたびに格納する．これにより，N 番目の情報を参照すると，ソースコード命令が N 回実行されたときの状況を表示できる．

復元時には，配列の宣言時に格納したアドレスと要素へのアクセス時に格納したアドレスを調べることで，配列の要素を表示する際，アドレスでなく名前を表示させることが可能である．また，ロック・アンロックの際も同様に名前の取得を行う．

3.4　表示

表示の際の工夫について，この章のはじめで説明していない部分について説明する．行った工夫は以下の通りである．

- 実行中のスレッド情報の表示

  図 3.3 の上部には，表示している地点で実行中のスレッド情報が表示される．ここでは，スレッド名・状態・実行中のソースコード上の行番号が表示されている．

- タブの色分け

  表示している地点で存在するスレッドの状態が一目で分かるように，色分けを行っている．行った色分けは，その地点で実行中のスレッドは赤，他のスレッドのうち実行待機中のスレッドは黒，ブロックされたスレッドは灰色としている．

- 表示する変数の項目による色分け

表示している各変数情報には，型・名前・値の項目がある．これらの項目をただ並べて表示するだけでなく，型は赤，名前は青，値は薄い紫というように色分けすることで情報の誤認を防ぐ．

- ロック情報の表示

  グローバル変数・フィールド変数が他のスレッドによってロックされている場合，ロックしたスレッド名を表示する (図 3.5)．この情報はスレッドがロックされると表示され，アンロックされると元の表示に戻す．

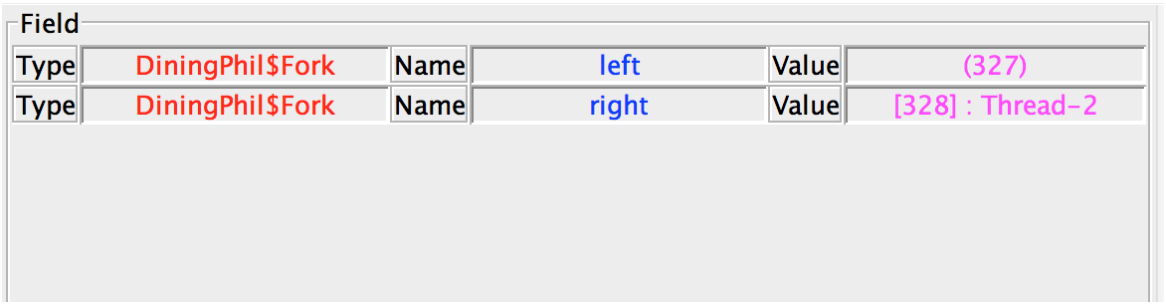

図 3.5　ロック時の表示

第 4 章

評価

4.1 実行時間の評価

本研究で採用した実装方法では，探索中に情報を格納する処理を行うため探索時間に影響を及ぼすと考えられる．そこで，本研究による拡張前と拡張後でどの程度探索時間に差が生じるのか検証を行った．検証には JPF 付属のサンプルプログラムである「哲学者の食事問題」を使用し，スレッド数を変更することで探索時間にどの程度影響を与えるのか比較した．また，実行環境の状態により探索時間が変化する可能性があるため，10 回探索を行ったときの平均時間を探索時間とした．

結果は表 4.1 の通りである．スレッド数によって多少の違いはあるものの，本研究による拡張後の探索時間は，未拡張時の約 1.25 倍かかることがわかった．この結果より，探索時間が数分から数時間程度のものであれば，探索時間の増加は許容範囲内であるといえる．

4.2 表示項目の評価

本研究で作成した Listener について，被験者に使用してもらうことで評価を行った．被験者は 5 人である．被験者により指摘された問題点と，それに対する解決策について以下に記述する．

- 検出されたエラーについての情報が表示されない．

  本研究による提示情報には，検出されたエラー情報が存在しない．この情報は JPF によって出力されるエラー情報を参照することで得られるが，提示情報として表示され

表 4.1　スレッド数変更時の探索時間

	Listener 未使用	Listener 使用	
スレッド数	平均探索時間	平均探索時間	増加率
15	59 秒	1 分 14 秒	1.254
16	2 分 8 秒	2 分 38 秒	1.234
17	4 分 29 秒	5 分 42 秒	1.271
18	9 分 36 秒	11 分 51 秒	1.234
19	20 分 22 秒	25 分 36 秒	1.257
20	42 分 33 秒	53 分 26 秒	1.256

るのが理想である．そこで，JPF によって出力される情報からエラーの種類がわかる程度の情報を表示させる．これは図 2.2 中の “java.lang.AssertionError” のような情報である．

- 表示項目について，何を表示しているのかわかりづらい．

  本研究による情報の提示では，それぞれの表示項目の説明は表示していない．これは表示情報が多くなると見づらくなるためであり，表示方法をシンプルにすることで各表示項目が何の情報であるかわかりやすくしたつもりだが，不十分であった．この問題は，各項目上にカーソルをあわせたときのみ説明を表示することで解決されると考えられる．この方法では各表示項目の説明が知りたい場合のみ情報を提示するため，見やすさに影響を与えない．

- 変更があった変数情報がわかりにくいため強調等の工夫が欲しい．

  この問題点については，被験者の意見通り変更があった変数情報を強調表示する．具体的には変数情報を表示している項目の色を反転させることで解決されると考えられる．

- オブジェクト型の変数について，名前があるものはアドレスではなく名前で表記して欲しい (哲学者の食事問題の Fork)．

  評価のために使用した「哲学者の食事問題」のプログラムでは，メインスレッドで生

成した Fork という名前の配列の要素を各スレッドに割り当てる．提示情報では，各スレッドに割り当てられた Fork は “Fork[?]” という表示ではなく，アドレスで表示されるためわかりづらいという問題である．この問題は，復元処理の際，他のスレッドに変数を割り当てることを想定していなかったためであり，アドレスと名前の対応付けを全スレッド共通で行うことで解決されると考えられる．

- 実行中のスレッドのタブ名はすべて赤で表示されるため，実行中のスレッドの状態変化がわかりづらい．

  本研究による提示では，実行中のスレッド名は赤，実行待機中のスレッド名は黒，実行をブロックされているスレッド名は灰のように，スレッドの状態によって色分けを行っている．しかし，実行中のスレッドは状態にかかわらずすべて赤で表示される．この問題の解決策としては，実行中のスレッド名の色を赤系統の色で色分けをすることで解決されると考えられる．

この結果より，表示方法について多くの改良が必要であることが明らかになった．

第 5 章

まとめ

本研究では，JPF による探索でエラーを検出した際，より詳細なトレース出力を提示する方法を提案した．本研究により，エラー検出までの経路中の情報の表示・閲覧が可能となった．しかし，被験者による評価から，検出したエラー情報の追加表示や変更のあった変数情報の強調表示，表示項目自体の説明の追加など，多くの改善が必要であることがわかった．今後の課題として，指摘された問題点を改善し，提案手法を用いることで利用者の作業負担がどの程度軽減されるのか被験者実験により明らかにしたい．

謝辞

本研究を進めるにあたり，高田喜朗准教授には多くの御指導・御指摘をいただきました．この場を借りて厚く御礼申し上げます．

また，副査を引き受けていただいた松崎公紀准教授, 酒居敬一講師ならびにお世話になった教員の方々に深く御礼申し上げます．

参考文献

[1] 中島 震, “SPIN モデル検査”, 近代科学社, pp2–32, 2009.

[2] 磯部 祥尚, 粂野 文洋, 櫻庭 健年, 田口 研治, 田原 康之,“ソフトウェア科学基礎—最先端のソフトウェア開発に求められる数理的基礎”, 近代科学社, pp6,7,298–305, 2008.

[3] “Java Path Finder”, http://babelfish.arc.nasa.gov/trac/jpf/wiki

[4] “Java PathFinder - SourceForge”, http://javapathfinder.sourceforge.net/

[5] 横山 翔一： Java PathFinder を用いたモデル検査における GUI の拡張, 北海道大学卒業論文要旨, 2007.

[6] “Eclipse による Java アプリケーションのデバッグ”,
http://www.oki-osk.jp/esc/eclipse3/eclipse-debug.html#basic9

[7] “Java Pathfinder の機能拡張”,
http://hagi.is.s.u-tokyo.ac.jp/pub/staff/hagiya/kougiroku/jpf/is090629.pdf

[8] “Java Pathfinder - PLG - University of Waterloo”,
http://plg.uwaterloo.ca/ olhotak/seminars/PParizek-JPF.pdf

付録 A

トレース情報設定時の実行結果

```
====================================================== trace #1
------------------------------------------------------ transition #0 th
read: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"<root>" ,1/1,isCascade
d:false}
      [2895 insn w/o sources]
  SumCheck.java:1                : public class SumCheck {
      [2 insn w/o sources]
  SumCheck.java:1                : public class SumCheck {
  SumCheck.java:2                : static int sum = 0;
      [2 insn w/o sources]
  SumCheck.java:13               : ThreadSum[] s = new ThreadSum[3];
  SumCheck.java:15               : for(int i = 0; i < 3; i++) {
  SumCheck.java:16               : s[i] = new ThreadSum();
  SumCheck.java:5                : ThreadSum() {
      [188 insn w/o sources]
  SumCheck.java:6                : }
  SumCheck.java:16               : s[i] = new ThreadSum();
  SumCheck.java:17               : s[i].start();
      [1 insn w/o sources]
------------------------------------------------------ transition #1 th
read: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"start" ,1/2,isCascaded
:false}
      [2 insn w/o sources]
  SumCheck.java:15               : for(int i = 0; i < 3; i++) {
  SumCheck.java:16               : s[i] = new ThreadSum();
  SumCheck.java:5                : ThreadSum() {
      [141 insn w/o sources]
  SumCheck.java:6                : }
  SumCheck.java:16               : s[i] = new ThreadSum();
  SumCheck.java:17               : s[i].start();
      [1 insn w/o sources]
------------------------------------------------------ transition #2 th
read: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"start" ,1/3,isCascaded
:false}
      [2 insn w/o sources]
  SumCheck.java:15               : for(int i = 0; i < 3; i++) {
  SumCheck.java:16               : s[i] = new ThreadSum();
  SumCheck.java:5                : ThreadSum() {
      [141 insn w/o sources]
```

```
  SumCheck.java:6                : }
  SumCheck.java:16               : s[i] = new ThreadSum();
  SumCheck.java:17               : s[i].start();
      [1 insn w/o sources]
------------------------------------------------------ transition #3 th
read: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"start" ,1/4,isCascaded
:false}
      [2 insn w/o sources]
  SumCheck.java:15               : for(int i = 0; i < 3; i++) {
  SumCheck.java:19               : for(int i = 0; i < 3; i++) {
  SumCheck.java:20               : s[i].join();
      [1 insn w/o sources]
------------------------------------------------------ transition #4 th
read: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"wait" ,1/3,isCascaded:
false}
      [2 insn w/o sources]
------------------------------------------------------ transition #5 th
read: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"sharedField" ,2/3,isCa
scaded:false}
      [2 insn w/o sources]
------------------------------------------------------ transition #6 th
read: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"sharedField" ,1/3,isCa
scaded:false}
  SumCheck.java:8                : sum++;
------------------------------------------------------ transition #7 th
read: 1
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"sharedField" ,1/3,isCa
scaded:false}
  SumCheck.java:8                : sum++;
  SumCheck.java:9                : }
      [1 insn w/o sources]
------------------------------------------------------ transition #8 th
read: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"terminate" ,1/3,isCasc
aded:false}
      [2 insn w/o sources]
  SumCheck.java:19               : for(int i = 0; i < 3; i++) {
  SumCheck.java:20               : s[i].join();
      [1 insn w/o sources]
------------------------------------------------------ transition #9 th
read: 3
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"wait" ,2/2,isCascaded:
false}
      [2 insn w/o sources]
------------------------------------------------------ transition #10 t
hread: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"sharedField" ,1/2,isCa
scaded:false}
  SumCheck.java:8                : sum++;
------------------------------------------------------ transition #11 t
hread: 3
```

```
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"sharedField" ,2/2,isCa
scaded:false}
  SumCheck.java:8               : sum++;
------------------------------------------------------ transition #12 t
hread: 2
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"sharedField" ,1/2,isCa
scaded:false}
  SumCheck.java:8               : sum++;
  SumCheck.java:9               : }
      [1 insn w/o sources]
------------------------------------------------------ transition #13 t
hread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"terminate" ,1/2,isCasc
aded:false}
      [2 insn w/o sources]
  SumCheck.java:19              : for(int i = 0; i < 3; i++) {
  SumCheck.java:20              : s[i].join();
      [1 insn w/o sources]
------------------------------------------------------ transition #14 t
hread: 3
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"wait" ,1/1,isCascaded:
false}
  SumCheck.java:8               : sum++;
  SumCheck.java:9               : }
      [1 insn w/o sources]
------------------------------------------------------ transition #15 t
hread: 0
gov.nasa.jpf.jvm.choice.ThreadChoiceFromSet {id:"terminate" ,1/1,isCasc
aded:false}
      [2 insn w/o sources]
  SumCheck.java:19              : for(int i = 0; i < 3; i++) {
  SumCheck.java:22              : }catch(Exception e) {}
  SumCheck.java:23              : assert(sum == 3);
      [21 insn w/o sources]

====================================================== results
error #1: gov.nasa.jpf.jvm.NoUncaughtExceptionsProperty "java.lang.Asse
rtionError  at SumCheck.main(SumChec..."

====================================================== statistics
elapsed time:       00:00:00
states:             new=29, visited=9, backtracked=22, end=3
search:             maxDepth=16, constraints hit=0
choice generators:  thread=28 (signal=0, lock=1, shared ref=6), data=0
heap:               new=371, released=74, max live=351, gc-cycles=31
instructions:       3781
max memory:         81MB
loaded code:        classes=83, methods=1150

====================================================== search finished:
 13/01/31 12:43
============================================ search finished:
```