モデル検査技術を利用したプログラム解析器の生成ツール

山岡裕司[†] 胡振江[†,††]
武市正人[†] 小川瑞史[†,††]

本論文では，時相論理式によって仕様が記述されたプログラム解析をモデル検査技術の利用によって行うツールについて述べる．本ツールは，対象プログラム言語は Jimple，プログラム解析の仕様記述言語は時相論理 CTL-FV である．Jimple は Java と相互変換可能な 3 番地コードからなる中間言語であり，Java に比べプログラム解析や最適化が適用しやすい．また，CTL-FV は CTL（Computation Tree Logic）を拡張した時相論理であり，プログラム中の情報を述語に引用することを許したところに大きな特徴がある．CTL-FV によって多くのプログラム解析が記述できるため，本ツールを使用すると Jimple プログラムに対し様々な解析を自動的に行うことができるようになる．今回，モデル検査を既存のモデル検査ツール SMV をそのまま利用することによって実装が非常に簡単になり，Java 言語で約 500 行（コメント除く）のプログラムでこれが実現できた．また，標準ライブラリのいくつかのクラスに対して無用命令の検出を本ツールにより実行したところ，比較的大きなサイズのクラスに対しても数分で解析することができた．ここでは，主に本ツールの設計と実装について説明する．

Generation of Program Analyzer Based on Model Checking

Yuji Yamaoka,[†] Zhenjiang Hu,[†,††] Masato Takeichi[†]
and Mizuhito Ogawa[†,††]

In this paper, we describe a tool that automatically performs program analysis using model-checking techniques. The tool has two characteristics; the target program is Jimple, and the specification of program analysis is described in temporal logic CTL-FV. Jimple is mutually convertible with Java; it is a 3-address intermediate language and is easier to perform program analyses and optimizations than Java. CTL-FV is an extension of CTL (Computation Tree Logic) to allow quoting information in a program to formulas. CTL-FV can describe many program analyses, thus our tool can carry out various analyses automatically. By the use of the well-developed model-checker SMV, we implemented this tool with only 500 lines of code in Java. As an example, *dead code detection* is performed to some classes in the standard library, and relatively large classes can be analyzed in a few minutes. We explain the design and the implementation of the tool.

1. 序　　論

コントロールフロー解析やデータフロー解析といったプログラム解析は伝統的なプログラム最適化において必要不可欠な基本的な解析である．プログラム最適化はそれら基本的な解析の結果を用いて，適切なプログラム変換を適用する．たとえば無用命令の検出や定数伝播解析などはその例である[1),7)]．これらの解析は手動で実装したもので行われるのが普通だが，特に以下のような理由で意図するプログラム解析器がツールによって自動生成されるときの利益は大きい．

- プログラム解析の仕様はほとんどの文書で自然言語によって記述されている．それを手動で正しく実装するのは容易ではない．
- プログラム最適化においては次々とヒューリスティクスによるアルゴリズムが考案され，実装されいる．最適化効果を高めるため新たなアルゴリズムを考案したとき，その効果について手軽に実験できる．

このような背景のもとで，プログラム解析とモデル検査の親和性が指摘され[9)]，モデル検査技術を利用することによって自動的に意図するプログラム解析器が生成される可能性について論じられるようになってきた．

モデル検査とはシステムがある仕様を満たしている

† 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo
†† 科学技術振興事業団
Japan Science and Technology Corporation

かどうかを検証する形式的検証法の 1 つであり[2]，システムの有限状態モデルとそのシステムが満たすべき仕様として記述された時相論理式を与えられたとき，システムがその仕様を満たしているかどうかを網羅的に検査する．この方法の最大の特徴は十分な計算資源さえあれば全自動的に検証が行われるということである．高度な数学的な知識などが必要とされないうえ，専門家たちによって開発された SMV[6] や SPIN[4] などのモデル検査ツールが無償で利用できるなどの手軽さもあって，ハードウェアの設計段階での検証などに積極的に利用され発展してきた．そして計算資源が充実し，モデル検査技術がさらに発展するにともなって，モデルがとる状態空間が非常に大きくなるなどの問題があって難しいとされてきたソフトウェアの検証にもモデル検査が利用され始めた．

モデル検査技術をプログラム解析に利用するには，プログラム解析の仕様が時相論理で記述される必要がある．近年，時相論理 CTL に自由変数を導入した CTL-FV によって多くのプログラム解析が自然に記述できることが Lacey ら[5] によって示された．CTL-FV の特徴は対象プログラム中の情報を論理式中の述語に引用可能とした自由変数を導入したところにあり，それによってプログラム解析が簡潔にかつ美しく記述できるようになった．しかし，CTL-FV のような自由変数を持つ時相論理は既存のモデル検査ツールにそのままでは受け入れられないという問題点があった．

本論文では，時相論理 CTL-FV によって記述されたプログラム解析をモデル検査技術の利用によって自動的に行うツールの実装について説明し，例を用いてその有用性などについて考察する．本ツールは Java と相互変換可能な Jimple を対象プログラムとしているため，実用的なプログラムを解析対象とすることができる．さらに，既存のモデル検査ツール SMV を利用することなどで 500 行程度の Java プログラムで簡潔に実現できた．このような簡潔さはツールの拡張や管理を容易にする．

本論文の以下は次のように構成される．2 章ではその生成ツールの設計について述べる．3 章ではプログラム解析の一種である無用命令の検出を例として用いながら実装の詳細について述べる．4 章では Java のクラスライブラリ java.math.*の 5 つのクラスについて本ツールを用いて実際にプログラム解析を行い，その実行時間をもとにツールの有効性について考察する．5 章では主にプログラム解析器の生成という視点での関連研究について述べる．そして 6 章で結論を述べる．

2. 設　　計

この章では，我々が開発したプログラム解析器生成ツールの設計方針について述べる．

我々は本ツールを以下の方針で作成することにした．

- 仕様が時相論理で記述されたプログラム解析をモデル検査技術の利用によって全自動で行う．
- 解析対象のプログラミング言語は現在広く使用されている実用的なものとする．
- 時相論理は，プログラム解析の仕様を記述するのに有用なものとする．
- 既存のツールを組み合わせた簡潔な実装法でも，ある程度大きなプログラムを対象とすることができることを示す．

この目的に適合するように設計した本ツールの全体図を**図 1** に示す．本ツールは，解析対象プログラムとして Jimple，解析仕様記述言語として CTL-FV を受け付ける．図 1 において大きな四角で囲まれた部分が本ツールにあたる．本ツールは互いに非依存である，解析対象となる Jimple プログラムと解析の仕様記述である時相論理 CTL-FV 式を入力として受け付ける．そしてその仕様で記述された解析をモデル検査によって行えるようなモデルを対象プログラムから作成し，モデル検査ツールの入力言語を生成する．この際，対象プログラムに非依存であった CTL-FV 式は対象プログラムに特化された CTL 式に変換される．そして最後にモデル検査ツールによる検証が自動的に行われ，その結果を出力する．モデル検査ツールによる検証が成功した場合はプログラム解析の結果何も検出されなかったことを意味する．また，モデル検査ツールによる検証が失敗した場合はその反例出力を解析することにより，プログラム解析の結果に対応する対象プログラム中の場所を出力する．

このように本ツールを使用すれば，解析対象 Jimple プログラムと解析仕様 CTL-FV 式を入力するだけでそのプログラムに対する仕様どおりのプログラム解析結果を得ることができる．

以下では本ツールで使用した言語やモデル検査ツールについて述べる．

2.1 対象言語：Jimple

Jimple とは Java クラスファイルと相互変換が可能な 3 番地コードの言語である[12]．Java バイトコードはスタックマシン上のコードでありそのままでは解析が難しいため，それと等価でより解析が簡単な Jimple をプログラム解析の対象言語とした．Jimple の主な命令文を**図 2** に示す．なお，図 2 以外の命令文は多

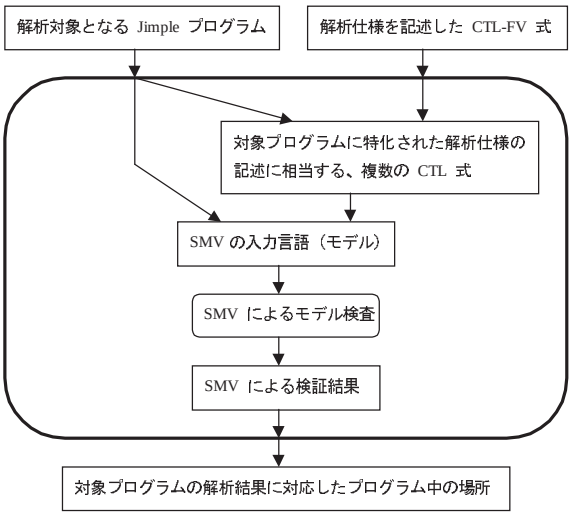

図 1 プログラム解析器生成ツールの全体図
Fig. 1 The whole picture of the generation of program analyzer.

```
〈代入文〉 ::=   〈変数名〉 = 〈式〉;
〈同値文〉 ::=   〈ローカル変数名〉 := @this : 〈this の型〉;
              | 〈ローカル変数名〉 := @parameter〈n〉 : 〈parameter〈n〉 の型〉;
              | 〈ローカル変数名〉 := @caughtexception;

〈ジャンプ文〉 ::=   goto 〈ラベル名〉;
〈条件分岐文〉 ::=   if 〈条件式〉 goto 〈ラベル名〉;

〈メソッド起動文〉 ::=   〈メソッド起動式〉;
〈メソッドからリターンする文〉 ::=   return;
〈メソッドから値をリターンする文〉 ::=   return 〈ローカル変数名〉;
                      | return 〈定数〉;

〈何もしない文〉 ::=   nop;
```

図 2 Jimple の主な命令文
Fig. 2 The main statements of Jimple.

分岐やモニタや例外に関係する命令文であり，ここでは説明を割愛した．

Jimple で記述されたプログラムはたとえば**図 3** のようになる．これは**図 4** の Java プログラムと等価で，そのクラスファイルと相互変換が可能なものである．このプログラムは $MyLib$ という名前のクラス定義であり，クラス $MyLib$ はデフォルトコンストラクタと static メソッド $int\ mss(int[])$ を持っている．なお，$mss(int[])$ は最大部分列和問題（Maximum Segment Sum，mss）を解くメソッドである．Java プログラムの変数 a，ret，$debug$，s，i はそれぞれ Jimple プログラムの変数 $r0$，$i0$，$i4$，$i1$，$i2$ に対応しており，Jimple プログラムのその他の変数は 3 番地コードに変換される際に導入されたものである．Jimple プログラムのラベル $label0$ 以前が Java プログラムの for 文の初期化文までに対応し，$label0$ と $label1$ と $label2$ で名付けられたブロックが Java プログラムの for 文の中身に対応し，同様に $label3$ と $label4$ が for 文および return 文に対応している．

2.2 プログラム解析の仕様記述：CTL-FV

CTL-FV は，分岐時間時相論理（branching-time temporal logic）の一種である CTL（Computation Tree Logic）に自由変数を引数に持つ述語を導入した時相論理である[5]．

プログラム解析の内容を時相論理によって簡潔に記述するためには，その論理体系にプログラムの状態を抽象化した述語の存在が不可欠である．手続き的なプログラミング言語で書かれたプログラムの状態で重要なものは主に変数の状態であるので，論理体系がプログラムの変数に対応付けられる述語を許容していることが望ましい．しかし一般に時相論理は命題論理であるためプログラム内の変数ごとに述語が必要となり，そのままでは記述が煩雑となる．そのために自由変数でプログラムの変数などを論理式中に引用する機能を付加した CTL-FV をプログラム解析の仕様記述言語とした．

たとえばプログラム中の無用命令の検出にあたる記述「プログラム中で，それ以降使用されることがない変数の値が定義されていることがない．」というも

```
class MyLib extends java.lang.Object
{

    private void <init>()
    {
        MyLib r0;

        r0 := @this: MyLib;
        specialinvoke r0.<java.lang.Object: void <init>()>();
        return;
    }

    public static int mss(int[] )
    {
        int[] r0;
        int i0, i1, i2, $i3, i4, $i5;
        boolean z0;

        r0 := @parameter0: int[];
        i0 = 0;
        z0 = 0;
        i1 = 0;
        i2 = 0;
        goto label4;

     label0:
        $i3 = r0[i2];
        i1 = i1 + $i3;
        if i1 >= 0 goto label1;

        i1 = 0;
        goto label3;

     label1:
        if i0 >= i1 goto label2;

        i0 = i1;
        goto label3;

     label2:
        i4 = i1;

     label3:
        i2 = i2 + 1;

     label4:
        $i5 = lengthof r0;
        if i2 < $i5 goto label0;

        return i0;
    }
}
```

図 3 図 4 と等価な Jimple プログラム
Fig. 3 A Jimple program equivalent to the Java program in Fig. 4.

```
class MyLib{
  private MyLib(){
  }
  public static int mss(int[] a){
    int ret=0;
    int debug=0;
    int s=0;
    for(int i=0;i<a.length;i++){
      s+=a[i];
      if(s<0)
        s=0;
      else if(ret<s)
        ret=s;
      else
        debug=s;
    }
    return ret;
  }
}
```

図 4 図 3 と等価な Java プログラム
Fig. 4 A Java program equivalent to the Jimple program in Fig. 3.

のを時相論理式で表現することを考えよう．その式を *dead-spec* と名付けると CTL-FV 式で，

dead-spec :
$\mathrm{AG}\,!(def(x)\,\&\,\mathrm{AX}\,!\mathrm{EF}\,use(x))$

などと表現することができる．しかしもし，自由変数を許容しない時相論理でこれを表現しようとすると，プログラム中のすべての変数について

$$
\begin{array}{lcl}
\phi & \in & \text{CTL-FV } \textit{proposition} \\
\phi & ::= & \textit{predicate}(x_1, \cdots, x_n) \\
 & | & !\phi \\
 & | & (\phi\,\&\,\phi) \\
 & | & (\phi\,|\,\phi) \\
 & | & (\phi \text{ -> } \phi) \\
 & | & \mathrm{E}\psi \\
 & | & \mathrm{A}\psi \\
 & | & \vec{\mathrm{E}}\psi \\
 & | & \vec{\mathrm{A}}\psi \\
 & & \\
\psi & ::= & \mathrm{F}\,\phi \\
 & | & \mathrm{G}\,\phi \\
 & | & \mathrm{X}\,\phi \\
 & | & [\phi\,\mathrm{U}\,\phi]
\end{array}
$$

図 5 CTL-FV の構文
Fig. 5 CTL-FV syntax.

AG !(def & AX !EF use) であることを表現する論理式を記述しなくてはならない．なお，式 *dead-spec* 中の $def(x)$ は自由変数 x を引数に持つ述語で「変数 x の値が定義されている場所で真，そうでない場所で偽」であることを表現するものであるとする．同様に $use(x)$ は「変数 x の値が使用されている場所で真，そうでない場所で偽」を表現している．論理式中で同じ名前で表現された 2 つの自由変数 x は，対象プログラム中の同一な変数に束縛されることになる．なお，これらの述語の意味をプログラム解析器生成ツールにどう組み込むかという問題が考えられるが，ここでは簡単のため本ツール内でそれらの意味は定義済みとした．

分岐時間時相論理である時相論理 CTL は経路限定子（path quantifier）と時相演算子（temporal operator）の組によって時相を表現する．経路限定子は経路の分岐構造についての記述に使用され，A，E の 2 種類がある．時相演算子は経路の性質についての記述に使用され，F，G，X，U の 4 種類がある．

CTL-FV の構文を**図 5** に示す．図中の記号!は否定，& は連言，| は選言，-> は条件法をそれぞれ表す．CTL-FV が CTL と異なる点は自由変数を持つ述語が導入されている点と $\vec{\mathrm{A}}$，$\vec{\mathrm{E}}$ という逆向きの経路限定子（path quantifier）が導入されている点である．逆向きの経路限定子も多くのプログラム解析を表現するのに役立つ．

経路限定子 A，E およびその逆向きの $\vec{\mathrm{A}}$，$\vec{\mathrm{E}}$ は結

果として状態式 ϕ を返すような経路式 ψ に関する演算子で，それぞれ，

- Aψ
  これ以降のすべての経路で ψ を満たせば真の状態.
- Eψ
  これ以降に ψ を満たす経路が存在すれば真の状態.
- $\vec{\mathrm{A}}\psi$
  ここからさかのぼるのすべての経路で ψ を満たせば真の状態.
- $\vec{\mathrm{E}}\psi$
  ここからさかのぼって ψ を満たす経路が存在すれば真の状態.

という意味を持つ.

また，時相演算子 F，G，X，U は結果として経路式 ψ を返すような状態式 ϕ に関する演算子で，それぞれ，

- F ϕ
  この経路上でいつか ϕ を満たせば真の経路.
- G ϕ
  この経路上でつねに ϕ を満たし続ければ真の経路.
- X ϕ
  この経路上の次の時間で ϕ を満たせば真の経路.
- $[\phi_1$ U $\phi_2]$
  この経路上でいつか ϕ_2 を満たしかつそれまで ϕ_1 を満たし続ければ真の経路.

という意味である.

なお，時相論理の記述力によってどれだけ多くのプログラム解析が表現できるか異なるわけだが，Lacey らは伝統的な最適化で利用されるほとんどのプログラム解析は CTL-FV によって記述することができると予想している．以下では，定数伝播，共通部分式の削除，ループ不変式の巻き上げ，無用命令の削除，の 4 つの最適化でそれぞれ利用されるプログラム解析の CTL-FV による仕様記述例をあげる.

まず CTL-FV 式中で用いるいくつかの述語について説明する.

- $conLit(x)$：他の述語で用いられている自由変数 x が定数であるような場所で真.
- $def(x)$：変数 x の値が定義されている場所で真.
- $defExp(x, e)$：変数 x の値が式 e による代入 $x = e$ によって定義されている場所で真.
- $defVar(x, y)$：変数 x の値が変数 y の値（または定数 y_{conLit}）による代入 $x = y$ によって定義されている場所で真.
- $node(n)$：場所を同定するラベルである n によって示された場所で真.
- $trans(e)$：式 e 中のすべての変数について，どの変数もその値が定義されていないような場所で真.
- $use(x)$：変数 x の値が使用されている場所で真.
- $useExp(e)$：式 e と同じ構造がある場所で真.

これらの述語を用いて，CTL-FV 式で以下のように解析仕様を与えることができる.

- 定数伝播（constant propagation）の解析
  「ある変数 x が定数 c として実行前にあらかじめ計算できる，という状態の検出」という解析仕様は，

  $$\mathrm{AG}\,!(use(x)\ \&\ \vec{\mathrm{A}}[!def(x)\ \mathrm{U}\ (defVar(x, c)\ \&\ conLit(c))]).$$

- 共通部分式（common subexpression）の検出
  「ある式 e がそこでの利用可能な式として変数 y としてすでに定義されている，という状態の検出」という解析仕様は，

  $$\mathrm{AG}\,!(useExp(e)\ \&\ \vec{\mathrm{A}}[(!def(y)\ \&\ trans(e))\ \mathrm{U}\ defExp(y, e)]).$$

- ループ不変（loop invariant）式の検出
  「ある式 e がループ不変である，という状態の検出」という解析仕様は，

  $$\mathrm{AG}\,!((node(n)\ \&\ useExp(e))\ \&\ \mathrm{E}[trans(e)\ \mathrm{U}\ node(n)]).$$

- 無用命令（dead code）の検出
  「任意の変数 x の値が定義された後 x の値が使用されることがない，という状態の検出」という解析仕様は，

  $$\mathrm{AG}\,!(def(x)\ \&\ \mathrm{AX}\,!\mathrm{EF}\ use(x)).$$

なお，すべての仕様記述において AG ! で全体が括られているのは,「プログラムがある性質を満たすことがない」という断言にすることによって，もしそのような性質がプログラムにあったときモデル検査ツールの反例出力によってその位置を確かめられるようにするためである.

2.3 モデル検査ツール：SMV

モデル検査の技術は既存の研究の蓄積が大きい．広く利用されているモデル検査ツールではその成果による非常に洗練されたアルゴリズムが実装されているため，モデル検査には既存のツール SMV をそのまま利用することにした.

以下 SMV の入出力について簡単に説明する．SMV の入力言語ではモデルと仕様の両方を同時に記述する．モデルはクリプケ構造（Kripke structure）の記述そのものであり，いわばそれぞれの状態が変数の値で決まるようにモデル化されたシステムの有限状態遷移グ

```
MODULE main
VAR
  request : {f, t};
  state : {ready, busy};
ASSIGN
  init(request) := f union t;
  next(request) := f union t;
  init(state) := ready;
  next(state) := case
    state = ready & request = t : busy;
    1 : ready union busy;
  esac;
SPEC
  AG(request = t -> AF state = busy)
```

図 6 SMV への入力記述の例
Fig. 6 An SMV program.

ラフをそのまま変換したような単純なものである．

図 6 に SMV の入力の簡単な例をあげる．まず，VAR ブロックではこのモデルで使用される変数の宣言とその定義域が定義される．例では *request* が f, t, *state* が *ready*, *busy* の 2 値のみをとる変数であることが宣言されている．そして，ASSIGN ブロックではこのモデルの状態遷移の記述がなされる．例ではまず，変数 *request* の初期値および任意の時点の状態での値が f, t のどちらでもありうることが記述されている．続いて *state* の初期値が *ready* であること，任意の状態にいるときの *state* の次の時点での状態が記述されており，「次の時点での状態は，現時点での状態が $state = ready$ かつ $request = t$ ならば $state := busy$ な状態に，それ以外ならば *state* は *busy* あるいは *ready* のどちらかの状態に遷移する．」という記述がなされている．最後に SPEC ブロックで，このモデルに対する仕様が CTL 式によって与えられる．この例での仕様は「このモデルは $request = t$ を満たす状態になるといつか必ず $state = busy$ を満たす状態になる．」という意味であり，このモデルは仕様を満たしていると予想される．

実際，これを SMV に入力すると，SMV は初期状態からたどりうるすべての状態において仕様が満たされていることを検査し，

specification AG (request = t -> AF state = busy) is true

という結果が出力される．もし，満たされないような仕様を入力として与えていた場合は，満たされない状態になるに至る初期状態からの状態遷移が 1 つ反例として出力される．

3. 実装の詳細

この章では，前章の設計方針に基づいて実装したプログラム解析器生成ツールの内部的な動作について述べる．まず概要を述べ，その後詳細な動作を具体的な例を用いて説明する．

本ツールは，解析対象の Jimple プログラムと解析の仕様を記述した CTL-FV 式を入力として与えられた後，以下の順に動作する．なお以下の (1) から (3) は，CTL-FV 式は SMV に受け入れられないため，SMV が受け入れる CTL 式に変換するステップである．(4) はこの実装で最も重要なモデル生成を行うステップであり，(5) は SMV による検証とその結果の出力に関するステップである．

(1) 自由変数の抽出
入力として与えられた CTL-FV 式をパーズし，自由変数の集合を得る．

(2) 自由変数の束縛
それらを対象プログラム中の具体的な情報に束縛させる．束縛の場合の数は幾通りかあるが，それぞれの場合について以降の手続きを実行する．

(3) CTL-FV 式から CTL 式への変換
前までのステップで今や自由変数が束縛された CTL-FV 式を CTL 式に変換する．

(4) モデルの生成
対象プログラムを抽象化して前のステップで得られた CTL 式を検証するのに十分なモデルを生成し，その CTL 式とともに SMV の入力言語にする．

(5) 検証結果の出力
SMV によってモデル検査をする．検証結果が失敗に終わったら SMV の反例出力をもとに，対象プログラム中のどこで，CTL-FV 式中の自由変数がどう束縛されたときに，検証に失敗したかを出力する．

これらの動作を以降では具体的な例に対応付けてより詳細に説明する．プログラム解析の仕様の例として先に式 *dead-spec* として紹介した，プログラムの最適化の際に行われるプログラム解析の一種である無用命令の検出を，また，このプログラム解析を適用する対象プログラムの例として先にあげた図 3 のプログラムを使用する．

なお，今回の実装では CTL-FV 式中の述語は $def(x)$ と $use(x)$ のみを対象としているので CTL-FV 式中の自由変数はすべてプログラム中の変数に束縛されることになる．

3.1 自由変数の抽出

入力として与えられた CTL-FV 式より自由変数の集合を得る．今，CTL-FV 式から自由変数の集合を得る操作を FV とすると，無用命令の検出を表現した式 *dead-spec*

AG !($def(x)$ & AX !EF $use(x)$)

に対して，

$FV(\textit{dead-spec}) = \{x\}$

となる．また，たとえば，定数伝播（constant propagation）の解析を表現した CTL-FV 式である，

$\textit{const-spec}$:
AG !($use(x)$ & $\vec{\mathrm{A}}$[!$def(x)$ U
($defVar(x,c)$ & $conLit(c)$)])

に対しては，

$FV(\textit{const-spec}) = \{c, x\}$

のようになる．

3.2 自由変数の束縛

CTL-FV 式中の自由変数を対象プログラム中の各メソッドの各変数に順番に束縛させ，それぞれの場合について 3.3 節以降のステップを実行できるようにする．

ここでは式 *dead-spec* を図 3 のプログラムに適用する例で説明する．この例では，前のステップで抽出される変数は x のみであるので，これをこのプログラムの各変数に順番に束縛し，解析していくことになる．

さて，まずこのプログラムには 2 つのメソッドがあることが分かる．まずは 1 つ目のメソッド $< init >$ ()（デフォルトコンストラクタ）について無用命令の検出を行う．$< init >$ () にはただ 1 つの変数 $r0$ があるので，これに x を束縛し，3.3 節以降のステップを実行することによって $< init >$ () の変数 $r0$ についての無用命令の検出を行うことによってこのメソッドの解析をすることができる．

続いて，もう 1 つのメソッド $mss()$ について無用命令の検出を行う．今度の場合は変数が複数あるので，それぞれの変数についての無用命令の検出を行う必要がある．そのため，まず x をメソッド $mss()$ の $r0$ に束縛し，$< init >$ () の場合と同様に解析を行う．それが終わったら，メソッド $mss()$ の $i0$ についての解析，続いて $i1$ についての解析というように順に解析し，$mss()$ すべての変数に x を束縛していくことによってすべての無用命令の検出を行う．

このように対象プログラム中のすべてのローカル変数について CTL-FV 式に相当する解析を行う．なお，今回の実装では intraprocedure な解析，つまりそれぞれの単一メソッド内だけで行う解析のみを対象とした．

3.3 CTL-FV 式から CTL 式への変換

CTL-FV はそのままではモデル検査ツールに受け入れられないので，自由変数が束縛された CTL-FV 式を CTL 式に変換する．CTL-FV 式中の述語を，そのプログラム中の具体的な変数となった引数によって特化し，引数のない述語へと変換することによってこれを実現する．

たとえば，式 *dead-spec* は前のステップによって自由変数 x が図 3 の $mss()$ の $i1$ に束縛されたときに次のような CTL 式に変換される．

AG !($def\_i1\_$ & AX !EF $use\_i1\_$)

CTL-FV 式の $def(x)$， $use(x)$ はその引数が $i1$ に特化されたことにより，$def\_i1\_$ は「$mss()$ 中の変数 $i1$ の値が定義されている場所で真，そうでない場所で偽」である述語，$use\_i1\_$ は「$mss()$ 中の変数 $i1$ の値が使用されている場所で真，そうでない場所で偽」である述語となる．

3.4 モデルの生成

対象プログラムを抽象化して前のステップで得られた CTL を検証するのに十分なモデルを生成する．

対象プログラムである Jimple は 3 番地コードなので各命令の粒度が小さい．すなわち Jimple の 1 命令がプログラムの状態を変化させる最小の単位と考えてよい．そこで生成するモデルの各状態にプログラムの各命令文を割り当てることで，ほとんどのプログラム解析を行うのに十分なモデルが生成されると考えた．

各状態は変数の値によって定められなくてはならないため，各命令に対応する状態をその命令を表現するような値で抽象化する必要がある．各状態が保持しなければならない情報は，検証しようとする CTL 式で用いられている述語の真偽を観測するためのものとその状態がプログラムのどの命令にあたるかというものだけなので，これらの情報を変数によって表現すればよい．

また，状態遷移については各状態がプログラムの命令文に対応していることからプログラムのコントロールフローをマッピングすればよい．ただし，条件分岐文に対応する状態からの遷移については，一般にはその条件式の値は実行時まで知ることはできないので，どの分岐先にも非決定的に遷移しうるという抽象化を行った．

このように生成されたモデルの例を**図 7** に示した．この図は図 3 のプログラムに対して，式 *dead-spec* の自由変数 x が $mss()$ の $i1$ に束縛されたものに相当する CTL 式，

AG !($def\_i1\_$ & AX !EF $use\_i1\_$)

を検証するときに生成するモデルの一部で，図 3 の $label0$ から $label3$ までに対応するものである．図中の四角い枠が枠外の括弧で示したプログラムの命令にそれぞれ対応した状態を示している．また，それぞれの状態をつなぐ矢印がプログラムのコントロールフローに相当する状態遷移を示している．各状態はその

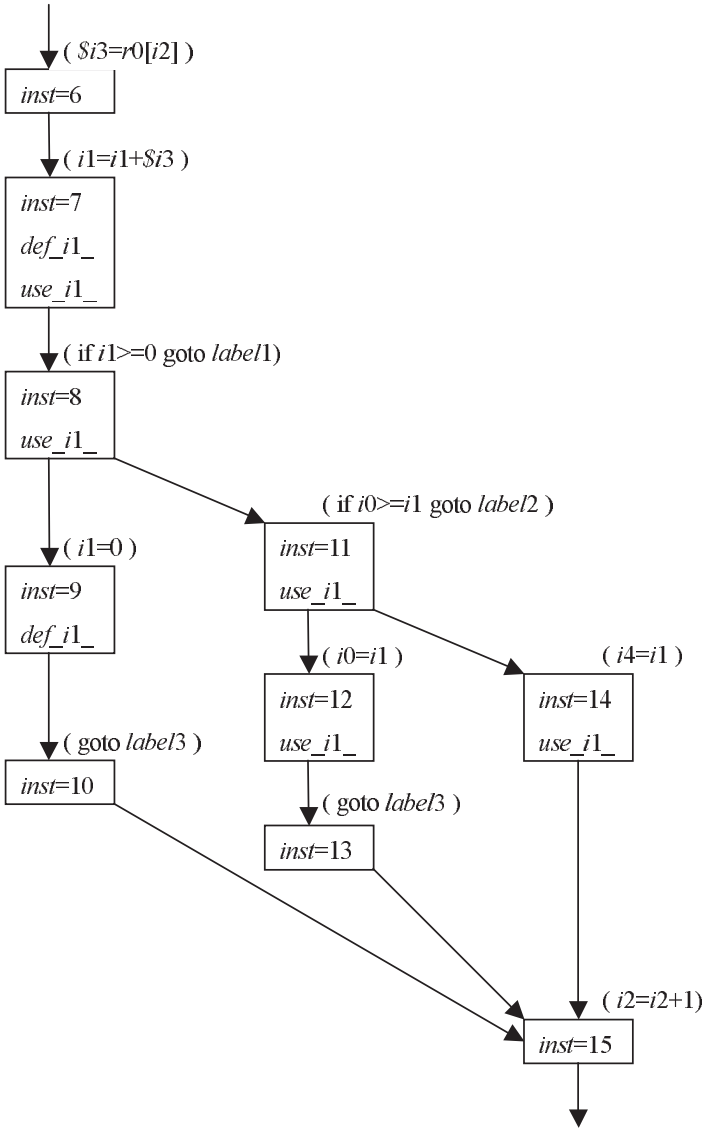

図 7 図 3 の *label*0 から *label*3 までに対応するモデル
Fig. 7 The model corresponding to *label*0 to *label*3 in Fig. 3.

メソッド中で第何番目の命令文か☆を表す変数 *inst* の値，$def\_i1\_$ および $use\_i1\_$ の情報を保持している．たとえば，プログラムの *label*1 の最初の命令文である if 文は，モデルである図 7 の中列の一番上の状態で示されている．この命令はメソッド $mss()$ 中で第 11 番目の命令であり，変数 $i1$ の値を使用している命令であるので，対応する状態は $inst = 11$ および $use\_i1\_$ を保持することになる．

そして最後に，以上のように生成されたモデルと CTL 式を SMV の入力言語に変換する．**図 8** に，式 *dead-spec* の自由変数 x が今度は $mss()$ の $i4$ に束縛されたときの SMV の入力言語を示す．図中の MODULE $NextInsts$ が *inst* の遷移を，$Predicate\_use\_i4\_$ がそれぞれの *inst* に対応する状態での $use\_i4\_$ の値を，$Predicate\_def\_i4\_$ がそれぞれの *inst* に対応する状態での $def\_i4\_$ の値を表現している．また，変数 *inst* は次の時点での状態に対応する値が代入されているものなので，変数 *output* は今の時点での状態に対応する値が代入されておりこれを結果表示の際に利用することになる．

なお，この方法は逆向きの経路限定子を含むような CTL-FV 式に対してはそのままでは使用することが

☆ メソッドの最初の命令文を第 0 番目の命令文とする．

```
MODULE NextInsts(inst)
DEFINE
  value :=
  case
    inst = 0 : 1;
    inst = 1 : 2;
    inst = 2 : 3;
    inst = 3 : 4;
    inst = 4 : 5;
    inst = 5 : 16;
    inst = 6 : 7;
    inst = 7 : 8;
    inst = 8 : 9 union 11;
    inst = 9 : 10;
    inst = 10 : 15;
    inst = 11 : 12 union 14;
    inst = 12 : 13;
    inst = 13 : 15;
    inst = 14 : 15;
    inst = 15 : 16;
    inst = 16 : 17;
    inst = 17 : 18 union 6;
    inst = 18 : -1;
    1 : -1;
  esac;

MODULE Predicate_use_i4_(inst)
DEFINE
  value :=
  case
    inst = 0 : 0;
    inst = 1 : 0;
    inst = 2 : 0;
    inst = 3 : 0;
    inst = 4 : 0;
    inst = 5 : 0;
    inst = 6 : 0;
    inst = 7 : 0;
    inst = 8 : 0;
    inst = 9 : 0;
    inst = 10 : 0;
    inst = 11 : 0;
    inst = 12 : 0;
    inst = 13 : 0;
    inst = 14 : 0;
    inst = 15 : 0;
    inst = 16 : 0;
    inst = 17 : 0;
    inst = 18 : 0;
    1 : 0;
  esac;
```

```
MODULE Predicate_def_i4_(inst)
DEFINE
  value :=
  case
    inst = 0 : 0;
    inst = 1 : 0;
    inst = 2 : 0;
    inst = 3 : 0;
    inst = 4 : 0;
    inst = 5 : 0;
    inst = 6 : 0;
    inst = 7 : 0;
    inst = 8 : 0;
    inst = 9 : 0;
    inst = 10 : 0;
    inst = 11 : 0;
    inst = 12 : 0;
    inst = 13 : 0;
    inst = 14 : 1;
    inst = 15 : 0;
    inst = 16 : 0;
    inst = 17 : 0;
    inst = 18 : 0;
    1 : 0;
  esac;

MODULE main
VAR
  output : {-1,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
  inst : {-1,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18};
  use_i4_ : {0,1};
  def_i4_ : {0,1};
  nextInsts : NextInsts(inst);
  module_use_i4_ : Predicate_use_i4_(inst);
  module_def_i4_ : Predicate_def_i4_(inst);

ASSIGN
  init(output) := -1;
  init(inst) := 0;
  init(use_i4_) := 0;
  init(def_i4_) := 0;

  next(output) := inst;
  next(inst) := nextInsts.value;
  next(use_i4_) := module_use_i4_.value;
  next(def_i4_) := module_def_i4_.value;

SPEC
  AG !(def_i4_=1  & AX !EF use_i4_=1 )
```

図 8 x が $i4$ に束縛されたときの SMV への入力
Fig. 8 The input to SMV when x is bound to $i4$.

できないが，式中の経路限定子がすべて逆向きである☆ような場合はモデルの状態遷移を反転させるだけで実現することができる．仕様記述中に順向きと逆向きの経路限定子が混在するようなプログラム解析は少ないので，実用上十分と考えている．

3.5 検証結果の出力

前のステップで生成したものを SMV によってモデル検査をする．検証に成功したときは，そのときの CTL-FV 式の変数の束縛では解析結果が true になったことに相当し，検証に失敗したときは，解析結果が SMV の反例出力として得られるのでそれを元のプログラムに対応付けて出力する．

以下では例として，再び図 3 のプログラムに対して，式 *dead-spec* の自由変数 x が $mss()$ の $i4$ に束縛された場合について説明する．この場合，もし検証に成功したら，メソッド $mss()$ の $i4$ についての無用命令は存在しなかったということであるので，(2) に戻って x を次の変数に束縛する．また，検証に失敗したら，それはそのモデル中の場所に対応する Jimple プログラム中の場所で CTL-FV 式 *dead-spec* が $i4$ について満たされていないということを意味している．

実際にはこの例の場合は SMV への入力は図 8 で

☆ 最外の AG！は除く．

```
— specification AG (!(def_i4_ = 1 & AX (!EF use_i4_ = 1)... is false
— as demonstrated by the following execution sequence
state 1.1:
output = -1
inst = 0
use_i4_ = -1
def_i4_ = -1
nextInsts.value = 1
module_use_i4_.value = 0
module_def_i4_.value = 0


...


state 1.13:
output = 11
inst = 14
nextInsts.value = 15
module_def_i4_.value = 1


state 1.14:
output = 14 ──▶ suggesting that the 14th Jimple statement is not verified.
inst = 15
def_i4_ = 1
nextInsts.value = 16
module_def_i4_.value = 0


resources used:
processor time: 0.05 s,
BDD nodes allocated: 4124
Bytes allocated: 1105908
BDD nodes representing transition relation: 168 + 1
```

図 9 モデル図 8 に対する SMV の反例出力の一部
Fig. 9 Part of SMV counterexample to the model described in Fig. 8.

あるので検証は失敗し，SMV により反例出力として**図 9** のように出力される．図の 1 行目に検証が失敗したことが明記されているのが分かる．それ以下の行で仕様が満たされなくなった状態にいたる遷移が表示されているが，興味があるのは仕様が満たされなくなった状態だけなので，そのときのプログラムの命令を示す情報である *output* の値を得ることによって解析結果が得られる．すなわち，元の Jimple プログラム図 3 の $mss()$ の $output = 14$ 番目の命令文，すなわち $label2$ の

$i4 = i1;$

が無用命令として検出された文であることが特定され，そのことがツールにより出力される．

以上，3.1 節から 3.5 節までのステップを経ることによってプログラム解析が自動的に行われる．

4. 実　　験

2 章で述べたように設計されたプログラム解析器生成ツールが，現実的な大きさのプログラムの解析を行うのにどれくらい時間がかかるか測定した．今回，Java により実装し Jimple のパーズには soot フレームワーク[12] という既存のフレームワークに含まれるライブラリを利用したため，約 500 行のソースコードを書くだけで実装することができた．

表 1 に Java1.3.1 のクラスライブラリ java.math.* の 5 つのクラスについて無用命令の検出にかかる時間の測定結果を示す．表には，対象プログラムのサイズおよびメソッド数，解析にかかった全時間，解析にかかった時間のうちモデル検査に費やされた時間を載せた．測定した環境は，OS が Windows98，Java ランタイムのバージョンは 1.3.1，CPU が 333 MHz Pentium II MMX でメインメモリは 128 MB である．この表から BigInteger のように比較的大きいサイズのクラスに対する解析も，このような遅い計算機上でも数分で解析できることが分かる．

なお，この表が示すようにこの実験によっていくつかの無用命令が検出された．たとえば，クラス BigInteger から以下のコードに無用命令が発見された．

```
int multpos = ebits;
ebits--;
boolean isone = true;
multpos = ebits - wbits;
```

このコードにおいて，1 行目の代入によって定義された *multpos* の値は使用される前に 4 行目によって

表 1 java.math.* の無用命令の検出にかかった時間
Table 1 Dead code detection of java.math.*.

クラス名	サイズ（bytes）	メソッド数	検出された無用命令の数	解析全体に使用された時間（sec）
SignedMutableBigInteger	1,146	8	0	1.81
BitSieve	1,948	10	0	5.77
BigDecimal	7,217	35	0	26.70
MutableBigInteger	12,966	50	11	167.96
BigInteger	28,888	104	14	346.58

表 2 java.math.BitSieve に対する解析にかかった時間
Table 2 Analysis times for java.math.BitSieve.

	CTL-FV 中の自由変数の数	全時間（sec）	モデル検査時間（sec）
AG !($def(x)$ & AX AG ($def(x)$ & $use(x)$))	1	6.32	1.97
AG !($def(x)$ & AX AG ($def(y)$ & $use(y)$))	2	106.17	52.46
AG !($def(x)$ & AX AG ($def(y)$ & $use(z)$))	3	7,137.79	3,953.41

再定義されているのが分かる．よってこの 1 行目の代入は無用命令である．これは Jimple 上での無用命令の検出の仕様として与えた式 *dead-spec* とは異なる仕様の無用命令であるが，このコードを変換した際 3 番地コードの Jimple では，

i13 = i2;
i39 = i2 + -1;
z0 = 1;
i40 = i39 - i1;

と変換され，$i13$ という二度と使われない変数が定義されてしまったために検出された無用命令である．なお，本ツールによってこれらのクラスのすべての無用命令が検出されたかどうかは確認していないが，検出された結果はすべて無用命令であったことは確認した．

また，**表 2** は先の実験と同様の環境で，サイズ 1,948 bytes,メソッド数 10 のクラス java.math.BitSieve に対して自由変数の数が異なる CTL-FV 式の解析を行ったときの測定結果である．3 種類の CTL-FV 式，

AG !($def(x)$ & AX AG ($def(x)$ & $use(x)$)) ,
AG !($def(x)$ & AX AG ($def(y)$ & $use(y)$)) ,
AG !($def(x)$ & AX AG ($def(y)$ & $use(z)$))

を仕様として入力し，それぞれについて測定した．これらの式はプログラム解析としての実用性はないが，モデル検査での検証が失敗することがほとんどないため，自由変数の数以外の条件に解析時間があまり影響されないと考えられる．表には CTL-FV 式中の自由変数の数，解析にかかった全時間，解析にかかった時間のうちモデル検査に費やされた時間を示した．3 章の実装では CTL-FV 式中の自由変数の数が増えると指数関数的に解析時間が長くなるはずであるが，この表からもそれが確認できる．そのことから CTL-FV 式中の自由変数の数が非常に多いときはこの方法は現実的でないが，通常プログラム解析の CTL-FV 式による仕様では自由変数が 5 つ以上になることは稀であり，より速い環境では十分実用的な手法であるといえよう．

5. 関 連 研 究

文献 5) はプログラム最適化を CTL-FV 式で記述された条件付きの書き換え系により記述することによって，その最適化がプログラムのセマンティクスを変更しないことを証明することができることを示した．そこでは，無用命令の削除（dead code elimination），定数の畳み込み（constant folding），ループ不変変数の巻き上げ（loop invariant hoisting）の 3 つのプログラム最適化の仕様を CTL-FV で与えており，CTL-FV がプログラム上の性質を簡潔に表現できることを示している．

プログラム解析器の自動生成についての研究には，たとえば文献 8) がある．これはプログラム解析の仕様を Cecil という言語によって記述することで，そのプログラム解析を自動化できるというものであり，本論文と非常に似ている．しかし，Cecil はプログラムの経路の性質を正規表現で記述するものであり，表現力は CTL-FV の E を除いたものに相当し，表現力に制限がある．また，その実装はプログラムを抽象化してモデルを得る部分と不動点計算を行うモデル検査に明確に分離していないため拡張性に欠けている．

また，プログラム解析にモデル検査技術を利用するアイデアは文献 9)〜11) を参考とした．文献 11) はコントロールフロー解析がモデル検査として見なせることを初めて指摘した．文献 9)，10) はさらに踏み込んで，抽象解釈がモデル生成に対応し，そのモデル上でモデル検査を行うことがデータフロー解析やコント

ロールフロー解析に対応していることを示している．

Java プログラム上でのモデル検査という研究については，Java プログラムのモデル検査による検証システム Bandera が文献 3) によって紹介されている．Bandera は対象プログラムに特化した具体的な仕様しか受け入れられないところが本論文の手法と大きく異なる．Bandera においても既存のモデル検査ツールをそのまま利用しているが，その入力となるモデル生成において Java プログラムから生成するモデルを縮小することについて力をいれている．この点においては今後おおいに参考になると考えている．また，今回は 2.1 節で述べた理由により解析の対象プログラムとして Jimple を用いたが，Bandera においても Jimple を利用している．Jimple はもともと McGill 大学で開発された Java バイトコードの最適化フレームワーク[12] において，プログラム解析および最適化用に利用される中間言語である．

6. 結　　論

本論文では時相論理によって仕様が記述されたプログラム解析器生成ツールの，設計と実装について説明した．用いた時相論理 CTL-FV は論理式中にプログラム中の情報を引用できるので，多くのプログラム解析を自然に記述することができる．また，対象プログラムは世の中で広く使用されている Java と相互変換可能な Jimple プログラムとした．これらの特徴により，本ツールは潜在的に多くの人が様々なプログラム解析を試みるのに役立つことが考えれらる．現在の実装は約 500 行の Java プログラムで書かれたものであり，実行効率向上の余地が多く残っているにもかかわらず実験によって現実的な大きさのプログラムに対する解析にも利用できることが確かめられた．このことはモデル検査技術を利用するプログラム解析の自動生成が，潜在的に大きな可能性を持っていることを示しているといえる．

現在検討中の，より効率的な実装については大きく 2 通りの方向性があると考えられる．1 つはより縮小されたモデルを生成する方向性である．与えられた仕様に対してモデルを特化し，モデル検査ツールが仕様を検証するに十分な大きさにできるだけモデルを縮小することによって，モデル検査の効率を劇的に向上させることができると考えられる．

もう 1 つは CTL-FV の自由変数とプログラム中の変数を束縛する機構をモデル検査ツールに組み込むといった，プログラム解析に特化したモデル検査ツールを開発する方向性である．このようなモデル検査ツールはモデルの状態を網羅的に検査しながら，与えられた仕様で検証すべき変数の束縛のみについて選択的に検証し，同一の仕様で何度も同じ状態を検査するようなことはしないだろう．ただ，2 章で述べたように既存のモデル検査ツールは非常に洗練されたアルゴリズムがいくつも実装されているので，これを拡張するのは注意が必要と思われる．

本ツールは対象プログラムが Jimple であったが，Java を Jimple に変換して本ツールに入力することによってクラスライブラリ中の無用命令をいくつか発見することができた．しかし，対象プログラムはあくまで 3 番地コードである Jimple としているため，たとえば Java プログラムのスタック操作に関する解析などは本ツールでは扱うことができないと考えられる．このことが Java プログラム解析に対して大きな障害となるかどうかまだよく分かっていない．

Java プログラム解析においては，解析結果を Java のソースコード上に反映したいという要求があると考えられる．前章で紹介した Java プログラムのモデル検査による検証システム Bandera は，Java ソースコードを Jimple に変換して検証を行い，その結果を元のソースコード上で表示するということに成功している．それと同等なテクニックを用いれば，本ツールによって得られた Java プログラムの解析結果をソースコード上で表示するということも可能だと思われる．

その他重要な課題として，時相論理 CTL-FV によって記述できるプログラム解析の性質の定式化についての研究もいまだ残っている．今後はこれらの課題を解決し，この手法でのプログラム解析の有効性をさらに主張していきたい．

参 考 文 献

1) Aho, A. and Ullman, J.: *Principles of Compiler Design*, Addison-Wesley (1977).
2) Clarke, E., Grumberg, O. and Peled, D.: *Model Checking*, MIT Press, Cambridge, Massachusetts (1999).
3) Corbett, J.C., Dwyer, M.B., Hatcliff, J., Laubach, S., Păsăreanu, C.S., Robby and Zheng, H.: Bandera: extracting finite-state models from Java source code, *International Conference on Software Engineering*, pp.439–448 (2000).
4) Holzmann, G.J.: The Model Checker SPIN, *Software Engineering*, Vol.23, No.5, pp.279–295 (1997).
5) Lacey, D., Jones, N.D., Wyk, E.V. and Frederiksen, C.C.: Proving correctness of compiler

optimizations by temporal logic, *Symposium on Principles of Programming Languages*, pp.283–294 (2002).
6) McMillan, K.: *Symbolic Model Checking*, Kluwer Academic Publishers (1993).
7) 中田育男：コンパイラの構成と最適化，朝倉書店 (1999).
8) Olender, K. and Osterweil, L.: Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation, *IEEE Trans. Softw. Eng.*, Vol.16, No.3, pp.268–280 (1990).
9) Schmidt, D.: Data Flow Analysis is Model Checking of Abstract Interpretations, *Proc. 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'98*, pp.38–48, ACM Press (1998).
10) Schmidt, D. and Steffen, B.: Program analysis as model checking of abstract interpretations, *Static Analysis*, Levi, G.(Ed.), Lecture Notes in Computer Science, Vol.1503, pp.351–380, Springer-Verlag (1998).
11) Steffen, B.: Data Flow Analysis as Model Checking, *Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science, Vol.526, pp.346–364, Springer-Verlag (1991).
12) Vallée-Rai, R., Co, P., Gagnon, E., Hendren, L., Lam, P. and Sundaresan, V.: Soot — a Java bytecode optimization framework (1999).

(平成 14 年 12 月 24 日受付)
(平成 15 年 7 月 1 日採録)

山岡　裕司

1978 年生．2001 年東京大学工学部機械情報工学科を卒業．2003 年東京大学大学院情報理工学系研究科数理情報学専攻修士課程修了．

胡　　振江（正会員）

1966 年生．1988 年中国上海交通大学計算機科学系を卒業．1996 年東京大学大学院工学系研究科情報工学専攻博士課程修了．同年日本学術振興会特別研究員を経て，1997 年東京大学大学院工学系研究科情報工学専攻助手，同年 10 月同専攻講師，2000 年同専攻助教授．2001 年より東京大学情報理工学系研究科助教授，同年 12 月より科学技術振興事業団さきがけ 21 研究者を兼任．博士（工学）．日本ソフトウェア科学会，ACM 各会員．

武市　正人（正会員）

1948 年生．1972 年東京大学工学部助手，講師，電気通信大学講師，助教授，東京大学工学部助教授を経て，1993 年東京大学大学院工学系研究科教授（情報処理工学講座），2001 年より同大学大学院情報理工学系研究科教授，現在に至る．工学博士．プログラミング言語，関数プログラミング，言語処理システムの研究・教育に従事．日本ソフトウェア科学会，日本応用数理学会，ACM 各会員．

小川　瑞史（正会員）

1985 年東京大学大学院理学系研究科数学専攻修士課程修了．2002 年博士（理学，東京大学・情報科学）．1985 年 NTT 武蔵野電気通信研究所入所．以来，関数型プログラムの解析・検証・生成，書き換え系の研究に従事．現在，科学技術振興事業団さきがけ 21 研究員および東京大学客員研究員．日本ソフトウェア科学会，ACM 各会員．