社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
SS2005-49 (2005-10)

Daikonの限定利用によるJavaメソッドの事後条件の自動導出

梶田 泰伸† 岡野 浩三† 楠本 真二†

† 大阪大学大学院情報科学研究科
〒560-8531 大阪府豊中市待兼山1-3
E-mail: †{y-kajita,okano,kusumoto}@ist.osaka-u.ac.jp

あらまし 近年Javaに対するDesign by Contract(DbC)の導入に関心が集まっている．事前・事後条件，不変表明をアサーションの形でJavaプログラムに記述することで，契約の取り決めを行うことができる．事前・事後条件，不変表明を表すアサーションを自動で生成するツールにDaikonがある．しかしDaikonはJavaプログラムの実行時の変数の値の情報をもとに，あらかじめ用意した性質のみをチェックしてアサーションを生成するため，変数間の関係が複雑である場合はあてはまる性質が存在せず単純なアサーションしか生成できない．そこで本稿ではこれらのアサーションの中でも事後条件に注目し，Daikonをループ部のみに限定して適用しその他の部分は変数をトレースして値を推論することにより，変数間の関係がより詳細な事後条件を生成する手法について述べる．この手法により，Daikonを単独で利用するより詳細な事後条件を生成することができた．

キーワード Design by Contract(DbC)，Java，事前条件，事後条件，Daikon，JML

Automatic generation of post-conditions for Java methods by limited utilization of Daikon

Yasunobu KAJITA†, Kozo OKANO†, and Shinji KUSUMOTO†

† Graduate School of Information Science and Technology, Osaka University
Machikane-yama 1-3, Toyonaka City, Osaka, 560-8531 Japan
E-mail: †{y-kajita,okano,kusumoto}@ist.osaka-u.ac.jp

Abstract Recently, the application of Design by Contract(DbC) to Java is concerned. The contracts can be made by describing pre-conditions, post-conditions, and invariants into the Java programs as assertions. Daikon is a tool which automatically generates assertions such as pre-conditions, post-conditions, and invariants. However, in generating assertions, Daikon checks only pre-defined properties based on the values of variables while running the Java programs. Thus when the relations between variables are complex, Daikon can generate only quite simple assertions because there are no pre-defined properties matching such those relations. In this paper, focusing on post-conditions, we propose a method which enables to generate detailed post-conditions with complex relations between variables. The method applies Daikon to only loop parts of the Java programs, while in the rest parts it traces the values of the whole variables. From a simple experiment, we found that the method generated more detailed post-conditions than Daikon alone.

Key words Design by Contract(DbC), Java, pre-condition, post-condition, Daikon, JML

1. はじめに

オブジェクト指向のソフトウェア設計に関する概念の1つにDesign by Contract(DbC)[1]がある．DbCではクラスとそのクラスを利用する側との間での仕様の取り決めを契約(contract)とみなすことにより，ソフトウェアの品質，信頼性，再利用性を向上を目指している．契約は，クラスの利用側がそのクラスを利用する際にある条件(事前条件)を保証すれば，そのクラスはある性質(事後条件)を満たすことを保証するというものである．またクラスは常に満たすべき性質(不変表明)も保証する．プログラムが契約に準拠するかどうかのチェックは，アサーションと呼ばれる真となるべき条件式を用いて行うことができる．アサーションはプログラムに直接挿入することができるため，プログラムの実行中に契約に違反した場合，直ちにそ

れを検出できる．また DbC に基づいて記述された事前・事後条件，不変表明は，コードそのものよりもプログラムの意味を端的に表しているという点や，コード中のコメント文よりも形式的に記述できるといった点から，プログラム理解を手助けするドキュメントとしても有効である．

DbC の概念に基づいたプログラミング言語には Eiffel [2] がある．Eiffel は高品質で信頼性のあるソフトウェアの生産を目的として設計されたオブジェクト指向のプログラミング言語である．Eiffel では，プログラム中に事前・事後条件，不変表明，その他のアサーションの記述が許されており，これらのアサーションの違反が起こると例外を発生させる機構が組み込まれている．

また Java に関しても DbC の導入が注目されている．J2SE 1.4 から assert 命令が導入され，プログラム内にアサーションを記述できるようになった．また，Java のための形式的仕様記述言語として JML(Java Modeling Language) [3], [4] がある．JML は仕様として事前・事後条件，不変表明を，Java コードに注釈の形で形式的に記述することができる．さらに JML の構文は Java とほぼ同じであるため．Java プログラマにとって親しみやすいものとなっている．その他の特徴としては，JML には\forall といった仕様記述を手助けする演算子が用意されており，極めて高い表現力を有している．また JML の仕様記述を利用した様々な検証ツールが存在し，実行時アサーションチェッキング，静的検査，形式的検証，モデルチェッキングといった，様々な検証が可能になる [5], [6].

しかし JML に限らず，大規模な Java プログラムに対して事前・事後条件，不変表明といった仕様を記述するには，多大な労力を費すことになる．そのため仕様の記述を支援するツールが重要である．そのようなツールの 1 つに，Daikon [7], [8] という Java プログラムから事前・事後条件，不変表明を表すアサーションを自動で生成するツールがある．Daikon は，Java プログラムの実行時に，プログラムポイントと呼ばれるメソッドの処理開始時，終了時といったプログラム中のある特定のポイントで変数の値を観察し，その情報をもとに Daikon があらかじめ用意している性質で満たすものを全て挙げ，アサーションとして生成する．しかし候補の性質が限られてしまうため，変数間の関係が複雑な場合にアサーションが生成できないといった問題がある．しかし現在，Daikon のこの問題の解決を目標とする研究は行われていない．

そこで本稿では，これらのアサーションの中でも事後条件に注目し，Daikon を Java プログラム全体に利用せずにメソッドの一部分のみに限定して利用し，それ以外の部分はメソッドを静的に解析することにより，変数間の関係がより詳細な事後条件を生成するツールを提案する．Daikon を適用するのはアサーションの生成が難しいと言われているループ部のみであり，他の部分は，メソッドの入口部から出口部まで変数をトレースして変数の値を推論することにより，変数の値に関する論理式を事後条件として出力する．変数のトレースはプログラムの処理の内容に沿って行うため，変数間の関係が複雑であってもプログラムの内容を反映した事後条件が出力される．そのため Daikon を単独で利用して生成された事後条件よりも，本ツールで生成した事後条件の方がプログラム理解や検証ツールへの適用の点において大きな効果が期待できる．また事後条件として出力された論理式は複雑になる可能性が高いので，ユーザが書き換えを行うことを想定して，書き換え前の論理式から書き換え後の論理式を導けるかどうかをチェックするような機構も提供する．

以降，2. で本ツールの内部でも使用する Daikon について詳しく述べる．次に 3. で本ツールについて述べ，4. で本ツールの実行例を示す最後に 5. でまとめと今後の課題について述べる．

2. Daikon

Daikon [7], [8] は事前・事後条件，不変表明を表すアサーションを自動で生成するツールである．プログラムを入力すると，まず入力されたプログラムに変数の値を観測するためのコードを追加する．そしてそのプログラムをユーザが別途用意したテストケース上で実行し，観測した変数の値からアサーションを推測する (図 1)．推測されたアサーションは，Daikon があらかじめ用意している性質の中で，観測した変数の値が反しないようなもの全てである．変数の値の観測は，プログラムポイントと呼ばれるメソッドの処理開始時，終了時といったプログラム中の Daikon が定めたポイントで独立に行われる．現在 Daikon は C 言語，Java，そして IOA [9] を対象としている．

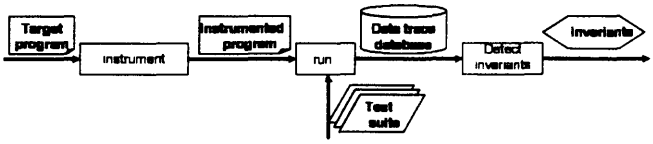

図 1　Daikon の処理の概要

例えば，スカラー変数 x, y, z と定数 a, b, c があるとき，変数の値が満たしているかどうかチェックされる性質の例としては，定数と等価であることを表す “$x = a$”，部分集合を表す “$x \in a, b, c$”，範囲を表す “$a \leqq x \leqq b$”，等がある．

Daikon は実行時の変数の値の情報のみからアサーションを推測するため，ブラックボックスであるようなプログラムにも適用できる．またプログラマが意図していない潜在的なアサーションを発見できるという点から，保守，デバッグ等にも有効である．

しかし，Daikon はあらかじめ用意した性質のチェックしか行わない．そのため変数間の関係が複雑になる場合は当てはまる性質が存在せず，単純なアサーションしか生成できないという問題がある．そこで Daikon の適用を，Daikon のアサーション生成手法が有効である範囲のみに限定し，その他の部分は別の手法を用いてアサーションの生成を行うことにより，より詳細なアサーションを生成することを考える．

3. Daikon の限定利用による事後条件生成ツール

本節では，アサーションの中でも事後条件に注目し，Daikon

をJavaプログラム全体に利用せずにメソッドの一部分のみに限定して利用し，それ以外の部分はメソッドを静的に解析することにより，変数間の関係がより詳細な事後条件を生成するツールについて説明する．

はじめに3.1で本ツールの概要を述べたのち，3.2で本ツールが内部での実質の解析対象であるJimpleについて説明する．次に3.3で本ツールが内部で行う主な処理について説明し，最後に3.4で生成された事後条件のチェック機構について説明する．

3.1 概　　要

本ツールは，入力されたJavaコードをJimple形式に変換した後解析を行い，各メソッドの事後条件を生成する．事後条件は，各メソッドに対してメソッドの入口から出口まで変数をトレースして変数の値を推論することにより導出する．アサーションの生成が難しいとされるループ部についてはDaikonを適用し，その結果を変数の値の推論に利用する．また生成された事後条件は複雑になる可能性が高いので，生成された事後条件に対してユーザが書き換えを行うことを想定して，書き換え前の論理式から書き換え後の論理式を導けるかどうかをチェックする機構も提供する．

本ツールの概要図を図2に示す．

現在，入力は1つのクラスのみ対応しており，分岐(if文)，ループ文(for文，while文)，同クラスのフィールド変数呼び出し，同クラスのメソッド呼び出しのみ解析が可能である．また出力は，同クラスのフィールド変数，及び解析したメソッドの返り値に関する論理式であり，論理式内で使う演算子，項等はJMLと同じである．

3.2 Jimple

本ツールはJavaを対象とした生成ツールであるが，入力されたJavaコードはSoot [10] を用いてJimple形式に変換し，SootのJimpleパーサを用いて解析する．SootはJavaコードの最適化フレームワークであり，Javaコードの解析や変換など目的に沿ったいくつかの中間表現を生成し，またその上での基本的な操作を行う環境を提供する．Jimpleは，Sootが生成する中間表現のうちの1つで，型付けされた3-address表現である．そのため変数の値の扱いや，解析するメソッド内のコントロールフローの抽出が容易であるといった理由で，Jimple形式への変換を行っている．

3.3 本ツールが内部で行う処理

本ツールの内部では，Jimple形式で表されたクラスに対して，Jimpleパーサを用いて構文解析を行ったのち，各メソッドに対して順に処理を行う．

以降で，本ツールが内部で行う処理で重要な部分を説明する．まず3.3.1で本ツールの基本アイデアである変数の値の推論について説明する．次に3.3.2でメソッド内にループが現れた際のDaikonの適用方法を説明する．最後に3.3.3で解析中のメソッド内に他のメソッド呼び出しが存在した際に行う処理について説明する．

3.3.1 変数の値の推論

解析対象のメソッドに対して，そのクラスのフィールド変数及びローカル変数の値を，メソッドの入口部の状態から出口の状態までトレースして推論することより，変数に関する事後条件を導出する．Jimpleコードにおいて式が書かれている1行を1状態と考え，状態0から始まるとする．

以下，簡単な例を使って説明する．図3はJavaメソッドの例であり，図4は図3のJavaコードをSootを用いて変換した結果出力されたJimpleコードである．また図4の“<数字>”は状態を表す．図4のJimpleコードでは状態0から状態7まで8状態ある．SootでJavaコードからJimple形式への変換を行うとこのような状態番号は現れない．説明の便宜上，付加している．

```
public int test(int x, int y){
    int ret;

    if(x > y){
        ret = x - y;
    }else{
        ret = y - x;
    }
    return(ret);
}
```

図3　Javaメソッドの例

```
 public int test(int, int)
 {
     IfTest this;
     int x, y, ret;

<0>  this := @this: IfTest;
<1>  x := @parameter0: int;
<2>  y := @parameter1: int;
<3>  if x <= y goto label0;

<4>  ret = x - y;
<5>  goto label1;

  label0:
<6>  ret = y - x;

  label1:
<7>  return ret;
 }
```

図4　図3と等価なJimpleメソッド

変数はx, y, retの3種類である．変数thisはクラスのインスタンスを表すものであり，this及びthisに関する状態である状態0は今回は無視する．次から各状態を順に解析し変数の値を推論していく．まず状態1の処理より，xにはメソッドの第一引数の値がセットされる．この場合Javaメソッドの引数の情報から第一引数はxであると分かるので，$x == \backslash\mathrm{old}(x)$となる．$\backslash$old()はメソッド呼び出し直前の値を表す．同様に，状態2では$y == \backslash\mathrm{old}(y)$となる．状態3ではif文による条件

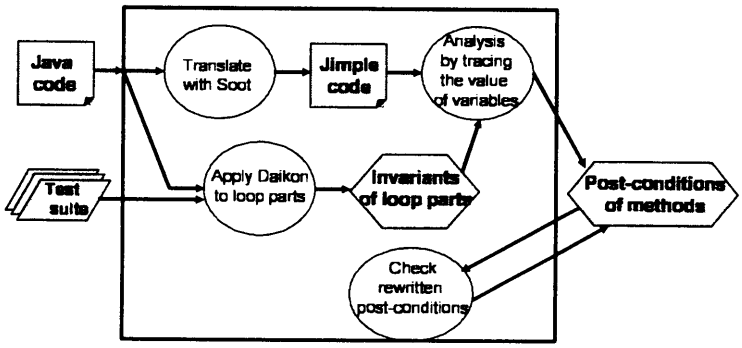

図 2　本ツールによる処理の概要

分岐があるため，goto 文の情報に基づき，状態 6 での処理により導出された論理式に条件文を，状態 4，状態 5 での処理により導出された論理式には条件文の否定を付け加える．条件文は $(x \leqq y)$ であり，この状態での x の値は $\backslash old(x)$ であり，y の値は $\backslash old(y)$ であることから，条件文は $(\backslash old(x) \leqq \backslash old(y))$ となる．状態 4 では ret への代入が行われている．状態 4 でも x，y の値はそれぞれ $\backslash old(x)$，$\backslash old(y)$ であることから，ret の値は $\backslash old(x) - \backslash old(y)$ にセットされる．さらに状態 3 で得た条件式の否定を付け加え，その結果 $\neg(\backslash old(x) \leqq \backslash old(y)) \Longrightarrow$ ret == $\backslash old(x)$ - $\backslash old(y)$ という論理式が得られる．"$\Longrightarrow$" は含意 (imply) を表す．次に状態 5 は何も行わず，状態 6 では，新たに $(\backslash old(x) \leqq \backslash old(y)) \Longrightarrow$ ret == $\backslash old(y)$ - $\backslash old(x)$ という論理式が得られる．そして状態 7 で返り値が ret であることが分かるので，ret に関する論理式を事後条件とする．その結果，このメソッドの事後条件として

$\neg(\backslash old(x) \leqq \backslash old(y)) \Longrightarrow \backslash result == \backslash old(x)$ - $\backslash old(y)$

$(\backslash old(x) \leqq \backslash old(y)) \Longrightarrow \backslash result == \backslash old(y)$ - $\backslash old(x)$

が得られる．\result は返り値を表す．

3.3.2　ループにおける Daikon の利用

解析対象のメソッドにループが含まれる場合，そのループ部にのみ Daikon を適用する．

Jimple コードからループ部を検出するには，ラベルに注目する．ラベルが，goto 文で参照されるよりあとの状態で宣言されていれば，それは通常の分岐であるが (図 5)，ラベルが，goto 文で参照されるより前の状態で宣言されていれば，それはループを表していると分かる (図 6)．図 5，図 6 の"<cond>" は条件式を表す．

```
   . . .
   if <cond> goto label0
   . . .
label0:
   . . .
```

図 5　通常の分岐

```
   . . .
label0:
   . . .
   if <cond> goto label0
   . . .
```

図 6　ループを表す分岐

メソッドの各状態を順に解析していく際に，このループ検出法によりループが検出されれば，ループ部に Daikon を適用する．Daikon は Jimple 形式には適用することはできないので，本ツールに入力された Java コードに対する適用を考える．しかし Daikon が生成するアサーションは事前・事後条件，不変表明のみで，メソッドの内部のある時点のアサーションというのは生成できない．そこで Java コードのループ部を抽出し，その部分をループと等価な操作を行うメソッド (ダミーメソッド) に置き換えて Daikon に適用することで，ループ部のアサーションを生成できるようにした．

図 7 はループを含む Java コードであり，図 8 は図 7 のループ部をダミーメソッドへの変換した後の Java コードである．

```
class A{
    public int loopTest(int i){
        int ret = 0;

        if(i<0){
            i = -i;
        }

        while(i>0){
            ret += i;
            i--;
        }
        return(ret);
    }
}
```

図 7　ループを含む Java コード

ここで，ループ部内で変数の値に変更が加わると，ダミーメソッド呼び出し後に続く処理にそれらの変更を反映させなければプログラムの意味が変わってしまう．そのためダミーメソッド内で利用する変数は全てフィールド変数とし，これらの変数はダミーメソッド呼び出し後に元の変数に値を戻している．

そして解析中のメソッドに対して Daikon を適用すると，解析中のメソッドに関するアサーションとともにダミーメソッドに関するアサーションが生成されるので，これをループ部のアサーションとする．そして引続き変数の値の推論を行う．

3.3.3　メソッド呼び出し

メソッド内に他のメソッドの呼び出しがある場合について説明する．現時点では，同じクラスのメソッドの呼び出しのみ解析可能である．

基本アイデアとしては，メソッド内に他のメソッドの呼び出

```
class A{
    int $d_i;    // dummy variable
    int $d_ret;  //

    public int loopTest(int i){
        int ret = 0;

        if(i<0){
            i = -i;
        }

        $d_i = i;       // copy to the dummy variable
        $d_ret = ret;   //
        $d_method01(); // call a dummy method
        i = $d_i;       // restore the value
                        //   to the original variable
        ret = $d_ret;   //

        return(ret);
    }

    // dummy method
    private void $d_method01(){
        while($d_i>0){
            $d_ret += $d_i;
            $d_i--;
        }
    }
}
```

図 8　ループ部をダミーメソッドに変換した後の Java コード

しがあれば，それまでの変数の値の推論と呼び出されたメソッドの事後条件を結合する．しかしそれまでの変数の値の推論で複雑な論理式が導かれている場合，呼び出されたメソッドの事後条件と結合するのは非常にコストがかかる．そこで，メソッド呼び出しまでに変数の値の推論により複雑な論理式が導かれている場合に限り，一時的な変数を用意しその変数に関する論理式を組み込むようにした．

以下，簡単な例で説明する．以下の図 9 のメソッド methodCall は，図 3 のメソッド test を呼び出している同じクラスのメソッドとする．

```
public int methodCall(int i){
    int j = 1;
    int ret;

    ret = this.test(i, j);

    return(ret);
}
```

図 9　図 3 のメソッド test を呼び出している同じクラスのメソッド

まずメソッド test の呼び出し直前の状態では，変数の値の推論により $i == \old(i)$，$j == 1$ という論理式が得られている．ここで，メソッド test は第一引数が x，第二引数が y であるため，メソッドの呼び出し test(i, j) の形から $\old(x)$ には i が，$\old(y)$ には j が対応する．よって，i，j に対応する一時的な変数 $t_i と $t_j を用意して，メソッド test の事後条件

$\neg(\old(x) \leqq \old(y)) \Longrightarrow \result == \old(x)$ - $\old(y)$

$(\old(x) \leqq \old(y)) \Longrightarrow \result == \old(y)$ - $\old(x)$

から，

$t_i == \old(i)$

$t_j == 1$

$\neg(t_i \leqq t_j) \Longrightarrow$ ret $==$ $t_i - $t_j

$(t_i \leqq t_j) \Longrightarrow$ ret $==$ $t_j - $t_i

を導出する．メソッド test の返り値は ret に代入されているため \result を ret に置き換えている．

この方法により，メソッド呼び出しまでに変数の値の推論で複雑な論理式が導かれていた場合でも対処することができる．

3.4　生成された事後条件の書き換えチェック機構

本ツールで生成される事後条件は複雑になる可能性が高い．そこで生成された論理式に対して本ツールのユーザが書き換えを行うことを想定して，書き換え前の論理式から書き換え後の論理式を導けるかどうかをチェックする機構を提供している．

この機構では，書き換え前の論理式を $E1, E2, ..., En$ とし，書き換え後の論理式を E としたときに，

$$E1 \wedge E2 \wedge ... \wedge En \Longrightarrow E$$

が成り立つかどうかを調べる．$E1, E2, ..., En, E$ に用いられる変数は現時点では int 型，long 型，short 型，float 型，double 型のみに限定している．また，得られた論理式も int 型しか用いられていないか，もしくは実数系の型しか用いられていないという制限も設ける．この制限により，前者に対してはプレスブルガー文真偽判定ルーチンを利用し，また後者については線形計画問題 (あるいは有理数上のプレスブルガー文判定問題) に帰着することにより判定可能となる．なお，$\old(x)$ などの出現はそのリテラルに対して 1 つの変数を与える (項表現に対しては自由解釈をする) こととし，一方，$<, >, =, +, -$ などの数式演算子はそのまま解釈することによりプレスブルガー文に置き換えている．

なお，整数型や浮動小数点型としてだけで表せない型を含むような式に対しては，項書換え系の利用，または全体として命題論理式に帰着可能な式のクラスについては同様に命題論理式の恒真性判定問題に帰着などの対処を考えている．

4. 実 行 例

本節では実行例について述べる．まず本ツールの実行結果との比較のために，Daikon のみを適用した結果について述べる．次に本ツールの実行結果を示し，最後にそれらについて考察を行う．本ツールは，事後条件の導出のために，Daikon の適用をループ部のみに限定してその他の部分は変数の値を推論を行うという点が重要である．そのためテストコードは図 7 で示した Java コードとした．

4.1 Daikon のみの実行結果

図 7 で示した Java コードに対して Daikon を適用した結果，以下の事後条件が生成された．なおテストケースは図 7 のメソッドの引数を境界値，及びランダムな数値に設定し，実行を十分であると考えられる回数繰り返した，

（1） (orig(arg0) == 0) $\Longrightarrow$ (return == 0)
（2） (return == 0) $\Longrightarrow$ (orig(arg0) == 0)
（3） return $\geqq$ orig(arg0)

orig はループ部に入る直前の値，return は返り値を表しており，3.3.1 で説明した \old()，\result() とそれぞれ意味は同じである．また arg0 はメソッドの第一引数を表す．

4.2 本ツールの実行結果

まず本ツールに図 7 の Java コードが入力されると Soot で Jimple 形式に変換され，解析を始めた．そして途中ループ検出法によりループを検出したため，入力された Java コードのループ部をダミーメソッドに変換し，入力されたテストケースで Daikon を実行する処理に移行した．なお現時点ではダミーメソッドは自動で生成できないため，この処理は手動で行った．またテストケースは 4.1 で用いたものと同じである．そしてその結果ループ部のアサーションが生成されたので，これを利用して以下の事後条件を生成した．

（1） (\old(i) $\geqq$ 0) && (\old(i) == 0) $\Longrightarrow$ \ result == 0
（2） (- \old(i) == 0) && $\neg$(\old(i) $\geqq$ 0) $\Longrightarrow$ \ result == 0
（3） (\old(i) $\geqq$ 0) $\Longrightarrow$ \result $\geqq$ \old(i)
（4） $\neg$(\old(i) $\geqq$ 0) $\Longrightarrow$ \result $\geqq -$\old(i)

4.3 考　　察

4.1 の (1) と 4.2 の (1) は同じ意味の論理式である．4.1 の (2) は，事前条件に関する論理式なのでここでは無視する．4.1 の (3) と 4.2 の (3),(4) を比べると，本ツールで導出された事後条件の方が詳細であることが分かる．

ただし本ツールの出力結果をみると (1) には冗長な表現が含まれていたり，(2) のような意味のない論理式が生成されたりと，本ツールの出力に若干の改良の余地があることがわかる．

5. おわりに

本稿では，Daikon を Java プログラム全体に利用せずにメソッドの一部分のみに限定して利用し，それ以外の部分はメソッドを静的に解析することにより，変数間の関係がより詳細であるような事後条件の生成を行うツールを提案した．本ツールでは，Daikon の適用はメソッドのループ部のみに限定し，その他の部分は変数をトレースし変数の値を推論を行うことにより，導き出された変数の値に関する論理式を事後条件として出力する．また生成された事後条件に対してユーザが書き換えを行うことを想定して，書き換え前の論理式から書き換え後の論理式を導けるかどうかをチェックする機構も提供している．

今後の課題としては，解析できる Java のクラス拡張が挙げられる．現在の実装では他クラスのインスタンスの参照は解析できない．これはクラス間の呼び出し関係を事前に保持しておかなければならず処理が複雑になる．その他の課題として，事前条件の生成も行いたいと思っている．現在検討中の案としては，Java メソッドの解析中に例外が投げられるような処理があれば，その時の変数の値の推論で導かれている論理式の否定を事前条件として生成することを考えている．また事前条件と事後条件を結合すれば不変表明も生成できると考えている．その他に，本ツールを適用して生成された事後条件の有効性の評価が課題である．本稿では単純な Java コードに本ツールを適用した結果，Daikon のみを適用した結果と比べて詳細な事後条件が生成されたことを示したにすぎない．評価は，仕様記述の大目的であるプログラム理解の点と，事後条件を利用した検証の点で，Daikon のみを適用した場合との比較を行いたいと思っている．また本ツールは出力する論理式の書き換えチェック機構を提供し，ユーザが論理式を書き換えることを前提としているが，それでも 4.2 で示したような冗長な表現や無意味な論理式は取り除きたいと思っている．その他にも，ダミーメソッド生成の自動化や，事後条件書き換えチェック機構の組み込みといった実装に関する課題も随時行っていきたい．

文　　献

[1] Bertrand Meyer. Applying "design by contract." Computer, 25(10):40-51, October 1992.
[2] Bertrand Meyer. Eiffel: The Language. Object-Oriented Series. Prentice Hall, NewYork, NY, 1992.
[3] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, Behavioral Specifications of Businesses and Systems, pp. 175 188, Kluwer Academic Publishers, Boston, 1999.
[4] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06u. Iowa State University, Department of Computer Science, April 2003.
[5] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino and E. Poll. An overview of JML tools and applications. FMICS'03, vol.80 of ENTCS, Elsevier, Trondheim, 2003
[6] Robby, Edwin Rodriguez, Matthew B. Dwyer, John Hatcliff. Checking Strong Specifications Using An Extensible Software Model Checking Framework. In the Proceedings of the Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004).
[7] Michael D. Ernst. Dynamically Discovering Likely Program Invariants. PhD thesis, University of Washington Department of Computer Science and Engineering, Seattle,Washington, August 2000.
[8] Michael D. Ernst, Jake Cockrell,William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. IEEE TSE, 27(2):1 25, February 2001. previous version appeared in ICSE, pp. 213-224, Los Angeles, CA, USA, May 1999.
[9] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. IOA: A language for specifying, programming, and validating distributed systems. Technical report, MIT Laboratory for Computer Science, 1997.
[10] Raja Vallee-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In Proceedings of CASCON 1999, pp. 125-135, 1999.