修 士 論 文

Javaの参照型変数と配列の静的null検出

平成 27 年度 修了
三重大学大学院　工学研究科
博士前期課程　情報工学専攻
コンピュータソフトウェア研究室

武田 真弥

概 要

Java バイトコードを静的解析し，NullPointerException の発生とその原因を検出する．静的解析は，対象のプログラムを実行せずに，コードを読み取ることで解析する．また，全ての経路を考慮した網羅的な解析が可能で，潜在的なエラーを検出できる．NullPointerException は Java で頻繁に発生するエラーで，null をデリファレンスしていると発生する．それが発生すると動作が停止してしまうなどの誤動作を引き起こす重大なエラーである．

FindBugs という Java の静的解析ツールがある．これは，NullPointerException を含んだ様々なバグを検出できるが，配列要素で発生する NullPointerException は解析できない．

また，配列を解析する Nikolic らのアルゴリズム (2012) は，配列と添え字に使われる変数に着目し，ループですべての要素が正しく初期化されるか解析する．

例 1: for(int i = 0 ; i < a.length ; i++) { a[i] = new A(); }

この例 1 のように，配列の添え字が 0 で始まり，比較，要素への代入，インクリメントがある場合に配列が正しく初期化できたと判定する．このアルゴリズムの問題点は，ループを用いた特定の初期化しか解析できず，各要素の初期化は解析できないことである．特に，初期化子を用いた初期化 (例: A a[] = {new A(), new A()}) が扱えない．

本研究では，このエラーを検出するために，参照型変数に null の代入がある箇所の集合を伝播することで解析する．経路の合流点では，その集合の和集合を求める．その集合が非空の変数は null になる経路があるとみなせるので，その変数をデリファレンスしていると警告を出す．また，集合の要素は null の代入箇所を示すので，その代入箇所と発生箇所との間の経路を求めることができる．

配列においては，配列の各要素を一つずつ初期化する場合とすべての要素を一度に初期化する場合に分けて解析する．各要素を一つずつ初期化する場合は，初期化された配列要素の番号を要素とする集合を用いて解析する．ループや API の Arrays.fill メソッドや toArray メソッドを用いて，すべての要素を一度に初期化する場合は，真偽値を用いて解析する．このように，配列の初期化を場合に分けて解析することで，集合の計算を減らすようにしている．

また，本研究は手続き間解析も行う．各メソッドに対してデリファレンスされる可能性のある仮引数番号の集合と返り値の要約情報を格納することで，引数と返り値に関係する NullPointerException を検出できる．一度解析されたメソッドの呼び出しがあると，そのメソッドの要約情報を用いるので，各メソッドは一度しか解析しない．

本研究の手法を，コンソールと統合開発環境 Eclipse のプラグインとして実行できるように実装した． バグの原因になる可能性が高い物のみを検出する． 実行時間は一万行程度のプログラムに対して一秒前後で解析できる．

目 次

第1章　はじめに

1.1　背景

Java とは 1995 年にサン・マイクロシステムズによりリリースされたオブジェクト指向プログラミング言語である．Java は様々なプラットフォームで動作することができる言語で，近年では Android アプリの開発言語としても需要が高まってきている．

Java ではプログラム実行中に起こる予期せぬエラーを例外という．プログラム実行の信頼性を高めるために，例外処理が実装されている．例外処理を記述している箇所で例外が発生すると，例外処理として任意の命令を実行させることができるが，それ以外の箇所で例外が発生するとプログラムは停止してしまう．NullPointerException(以下 null 例外) は Java プログラムの実行時に発生しやすい例外の一つである．これは参照型変数を値が null(参照先がない状態) でデリファレンス (フィールドアクセス) すると発生する例外である．null 例外は例外処理を書かなくてもコンパイルエラーにはならないので，プログラムが停止する大きな原因の一つである．

また，プログラムの開発において，滅多に通らない実行経路で発生するエラーを特定するのは，プログラムを動かすだけでは困難である．これにより，開発の最終段階でもバグに気づかず，リリース時にバグが残ってしまう恐れがある．そこで，潜在している null 例外を検出するために，実行可能な全ての経路が解析の対象であり，網羅的な解析ができる静的解析を用いる．静的解析は，解析対象のプログラムを実行することなく，コードを読み取ることで解析する．

1.2　目的

本研究の目的は，null 例外の発生を未然に防ぐため，Java バイトコードを静的解析し，参照型変数と配列の参照型要素，メソッド呼び出しの引数と返り値から発生する null 例外の箇所と原因を検出する手法の提案である．参照型変数による null 例外は，null の代入箇所と代入箇所から伝播する経路を特定できる手法で解析する．

また，提案した手法をツールの形として実装する．コンソールから起動できるように実装するとともに，利便性を高めるべく，統合開発環境 Eclipse のプラグインとして起動できるよう実装し，エディタ上で簡単に解析できるようにする．解析時間は一万行程度のプログラムなら，数秒で終わるよう短くする．ツールが示す警告はなるべく誤検出が少なくなるようにし，エラーになる可能性が高い物だけを表示する．

1.3 NullPointerException

本研究の解析対象である NullPointerException について詳しく解説する．

1.3.1 NullPointerException が発生する場合

Java Platform Standard Edition 7 ドキュメント [1] には以下のように記述されている．オブジェクトが必要な場合に，アプリケーションが null をデリファレンスするとスローされる．

1. null オブジェクトのインスタンスメソッドの呼び出し
2. null オブジェクトのフィールドに対するアクセスまたは変更
3. null の長さを配列であるかのように取得
4. null のスロットを配列であるかのようにアクセスまたは修正
5. null を Throwable 値であるかのようにスロー

1 と 2 と 3 は null なオブジェクトがドット演算子によりメンバを参照する場合，4 は null なオブジェクトの配列要素を指定した場合，5 は null な Throwable 型のオブジェクトを throw によってスローした場合にそれぞれ null 例外が発生する．その他にラッパークラスのアンボクシング機能による使用で発生する null 例外がある．

1.3.2 参照型変数が null を示す場合

- 参照型変数が宣言された時
  - 参照型変数の初期値は null
- null を示す参照型変数や返り値を参照 (代入) する場合
- 明示的に null を参照する場合

もし型に合わないインスタンスを参照すると ClassCastException が発生する．よって参照型変数が null でないなら，その参照型変数の型にあったインスタンス (非 null) を参照していることになる．

第2章　関連研究

2.1　静的解析ツール Findbugs[2]

Findbugs は Java の静的解析ツールである．Java のクラスファイルが入力で，null 例外のような動作が停止してしまう重大なバグから，動作には影響はないが推奨されていない記述まで幅広く検出する．Findbugs における，null 例外の検出手法に関する論文 [3]，[4] を紹介する．

この研究の目標は，誤検出が少なくなるように検出することである．null 例外の解析手法は，変数のデリファレンスがある箇所に着目し，そこからその値が null になるかどうかを後方解析する．また，アサーションを考慮したり，解析にアノテーションで指定した結果を用いることができる．

この研究の問題点としては，手続き内解析しかできない点であること，配列の要素が解析出来ない点が挙げられる．

2.2　Nikolic らの配列解析アルゴリズム [5]

オートマトン (図 2.1) を用いて配列の初期化状態を解析する．

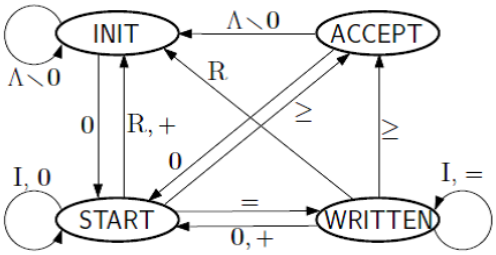

図 2.1: オートマトン

このオートマトンは各配列に対して用いられるもので，INIT から始まる．添字に使われる変数に 0 を代入するのを **0**，インクリメントするのを+，比較されるのを ≥，配列の要素に代入するのを=，配列または添字の変数にこれら以外の操作がされるのを **R**，配列または添字の変数に関係のない操作が **I** とする．ACCEPT になった配列が初期化されたとみなす．下記のループは配列が初期化されたとみなす．

```
for(int i = 0 ; i < a.length ; i++){
  a[i] = new A();
}
```

この解析は誤検出が起きないようになっている．ただし，検出漏れが多く，このようなループを用いた配列の初期化しか解析できないため，一つずつ配列要素に代入して初期化す

る方法などには適用できない．特に，初期化子を用いた初期化 (例: A a[] = { new A(), new A()}) が扱えない．

2.3 Madhavan らの手続き間解析アルゴリズム [6]

この研究は，Nanda らの研究 [7] を発展させたものである．クラスファイルが入力対象で，null 例外の発生に関する検証を行う手法の提案と実装を行う．

この研究の解析手法は，変数のデリファレンスがある箇所に着目し，最も成約のゆるい条件が一致するかを解析することで null 例外を検出する．つまり，少しでも怪しい物を検出する．手続き間解析も可能で，Nanda らの研究では扱えなかった再帰呼び出しにも対応している．他にもライブラリメソッドの呼び出しやフィールド変数も解析できる．

実行時間は 1 デリファレンスあたりの解析時間は瞬時に終わり，10 個のプログラムに対して評価を取ったところ，全てのデリファレンスの 16%のデリファレンスが安全でないと検出した．

この解析の問題点として，配列の要素が解析できないというのがある．

2.4 本研究の特徴

関連研究を踏まえ，本研究の特徴を述べる．

入力は関連研究と同じ Java クラスファイルが入力で，各メソッドを前方解析し，変数の値の伝播を解析する．参照型変数と関連研究では出来なかった配列の要素の解析ができる．引数と返り値を考慮できる手続き間解析ができるが，再帰呼び出しには対応していない．

危険性が高いものだけを警告し，実行時間は一万行程度のプログラムなら数秒で終わる．

第3章　準備

本研究の入力対象と，null 解析する前に必要な処理を示す．

3.1　入力対象

入力は JavaSE コンパイラで生成したデバッグ情報を含んだクラスファイルである．統合開発環境 Eclipse ではデフォルトで生成，JavaSE のコンパイラでは，-g オプションを付けてコンパイルすると生成される．デバッグ情報を付加しなかったり，別のコンパイラで生成されたクラスファイルでは正常に動作しない恐れがある．

クラスファイルには Java 仮想マシンで実行するための命令であるバイトコード，ローカル変数表，定数などを格納するためのコンスタントプールがある．クラスファイルを読み取り，Java のデータ構造として格納し，容易に扱えるようにするライブラリの BCEL を用いて解析する．

3.1.1　クラスファイル解析の利点

ソースコードを解析するのと比べ，クラスファイルを解析することはいくつかの利点がある．

1. バージョンアップによる言語仕様の変更に強い
2. クラスファイルしか提供されていないものが解析できる

1 は Java の言語仕様やコンパイラ (JDK) のバージョンが変わっても、生成されるバイトコードの命令は同じなので，クラスファイルが入力だと解析できる．2 はライブラリのようなクラスファイルしか提供されていないものでも解析できる．

今後の拡張次第では，クラスファイルを生成する別の言語 (Scala 等) を解析することも可能になる．また，クラスファイル解析のほうがソースコードを解析するよりデータの取得が容易，冗長なデータが少ないなどの点から，解析に都合の良いことが多い．

3.2 前処理

クラスファイルからバイトコードを読み取り，それを元にコントロールフローグラフを作成する．バイトコードの分岐命令 (例:if_eq) とジャンプ命令 (例:goto) に着目し，制御がそれらの命令で別の場所に移らない命令のまとまりを作る．そのまとまりをブロックと呼び，グラフの各ノードはそのブロックからなる．制御の移りを辺で繋ぐ (図 3.1)．また，クラスファイルからローカル変数とフィールド変数の情報も解析し，データ構造として表現する．

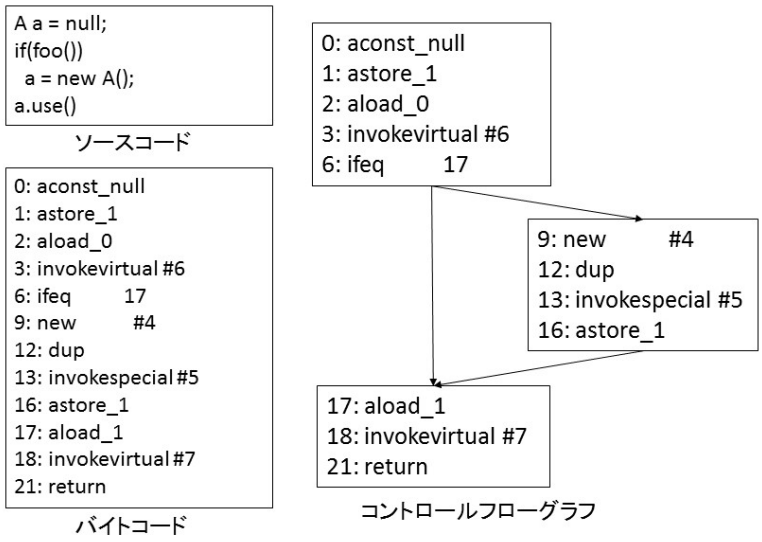

図 3.1: コントロールフローグラフの例

第4章　解析

4.1　解析の概要

参照型変数と手続き間解析の手法の概要を説明する．

4.1.1　参照型変数

本研究では参照型変数が null になる可能性があるかを解析する．プログラムをコントロールフローグラフで表し，各変数に対して用意された null 代入，非 null 代入，不明瞭代入のある場所を要素とする三つの集合の伝播を解析する．

図 4.1 は本研究の解析を簡略化し表現した図である．図の表は各ノードを解析後の変数の集合の状態を示す．null 代入があると，代入された変数の null 代入の集合をその位置にする．例えば，ノード A は変数 a と c に null を代入しているので，それぞれの集合はその位置を示す {A-1}，{A-3} になっている．また，非 null(new) が代入されると，その集合は空集合にする (例:B-1 の変数 a)．各ノードを解析するときに先行ノードがある場合は，各先行ノードの null 代入の集合の和集合を求める．例えば，ノード D の変数 c の null 代入の集合は，ノード B と C における変数 c の null 代入の集合の和集合を求めた物 {B-2, A-3} になる．

変数の null 代入の集合が非空のときは，要素が示す位置で代入された null 値がその解析地点まで到達することを示す．よって，デリファレンスされる変数の null 代入の集合が非空のときは null 例外が発生すると警告する．また，その要素は null 例外が発生する原因となる null 代入の箇所を示す．例えば，ノード D では変数 a，b，c がデリファレンスされ，null 代入の集合が非空な変数は b と c である．よって，D-2，D-3 で null 例外が発生すると警告し，それぞれの集合の要素が null 例外の原因となる null 代入の箇所である．

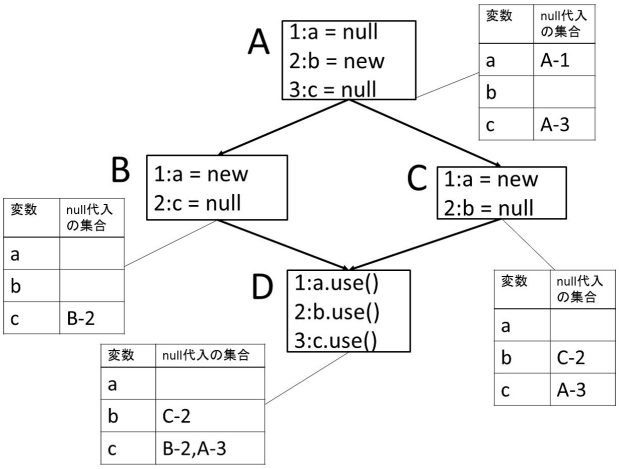

図 4.1: コントロールフローグラフと定義の状態

4.1.2 手続き間解析

本研究は手続き間解析をすることで，引数と返り値による null 例外を検出することができる．

メソッドごとに要約情報を持ち，メソッドの解析が終わると，その結果を要約情報として保持する．要約情報には，仮引数の値がデリファレンスされる可能性のある仮引数の一覧と，そのメソッドの返り値からなる．

あるメソッドを解析しているときにメソッド呼び出しがあると，そのメソッドの要約情報があるかを確認する．ない場合は呼び出し先メソッドを再帰的に解析する．それにより，呼び出し先メソッドの要約情報が生成されるので，その情報を用いて解析する．実引数が null になる可能性がある場合，それと対応する実引数の変数が呼び出し先の要約情報のデリファレンスされる可能性のある仮引数の一覧に含まれていると，危険なデリファレンスがあるとみなす．また，返り値の要約情報がある場合，それを解析に用いる．ただし，この手続間解析は各メソッドを一度しか解析しないので，メソッドが再帰している場合は上手く解析できない．

一度メソッド解析が終わると要約情報が生成され，次からはそのメソッドを解析することなく要約情報だけを利用する．よって，手続き間解析をしているが，すべてのメソッドを一回ずつ解析するのとほぼ計算量が変わらない．

・手続き間解析の例 1(引数による null 例外)

```
void foo(){
  A a = null;
  if(boo()){
    a = new A();
  }
  bar(a);//警告を出す
}
```

foo の要約情報:
デリファレンスされる仮引数:なし
返り値:なし

```
void bar(A para){
  A temp;
  if(boo()){
    temp = para;
  }
  para.use();//警告を出す
}
```

bar の要約情報:
デリファレンスされる仮引数:一つ目
返り値:なし

メソッド foo 内で null になる可能性のある変数 a がメソッド bar の実引数として使われ，その値がメソッド bar でデリファレンスしているために警告を出す．

・手続き間解析の例 **2**(返り値による **null** 例外)

```
void baz(){
  A a = get();
  a.use() ; //警告を出す
}
```

baz の要約情報:
デリファレンスされる仮引数:なし
返り値:なし

```
A get(){
  A temp = null;
  for(A v: ListA){
    if(v.boo()){
      temp = new A();
    }
  }
  return temp;
}
```

get の要約情報:
デリファレンスされる仮引数:なし
返り値:null になる可能性がある

get メソッドは null になる可能性がある変数 temp を返しているため，それを受け取る baz メソッドの変数 a のデリファレンスは警告を出す．これは，この例 2 のように，コレクションから条件に一致したものを返すときによく現れるコードである．

4.2 参照型変数の解析手法

参照型変数と手続き間解析についての解析手法を説明する．

4.2.1 解析の対象

解析手法を説明するために，参照型変数への代入，同じクラス内のメソッド呼び出し，return 操作，制御操作が行える部分言語を定める (表 4.1)．この言語では，Java の機能のフィールド変数への代入，継承，例外処理は扱わない．

名称	命令
NullAssign	$v = null$;
NonnullAssign	$v = nonnull$; //v = new;
ObsAssign	$v = obsvalue$;
VarAssign	$v = w$;
InvokeAssign	$v = innermethod(v1,v2,...vn)$;
Dereference	v.dereference;
Invoke	$innermethod(v1,v2,...vn)$;
Return	return v;
Etc	e;

表 4.1: 部分言語の命令一覧

上記の NullAssign-InvokeAssign を定義と呼ぶ．
NullAssign は null 値を代入する命令で，null 定義と呼ぶ．
NonnullAssign は Java では new 演算子のような，非 null を代入する命令で，非 null 定義と呼ぶ．
ObsAssign は解析対象外のフィールド変数やライブラリメソッドの返り値のような，解析できないため null か非 null かどちらになるかわからない値を代入する命令で，不明瞭定義と呼ぶ．
VarAssign は参照型変数 w を代入する命令である．
InvokeAssign はメソッド呼び出しの返り値を代入する命令で，*innermethod* は呼び出し先のメソッド，*v1,v2,...vn* は実引数の一覧を示す．返り値の要約情報 *innermethod.returnValue* の状態により，NullAssign，NonnullAssign，ObsAssign のいずれかの定義になると定める．
Dereference は変数 v をデリファレンスする命令で，この時の v が null とみなせる場合に警告を出す．
Invoke は InvokeAssign の変数への代入がないのと同じである．
Return は return 命令で，変数 v を返す．
Etc の e はいずれにも該当しない解析には関係のない命令で，変数宣言，制御操作がここに該当する．

プログラムの各命令をコントロールフローグラフのノードとし，その集合を N と表す．辺の集合を E=$\{(n,m) \mid n,m \in N$ かつ制御が n から m に移る $\}$ と定めると，コントロールフローグラフは両者の組 CFG=$\langle N,E \rangle$ で表せる．

解析に用いる構造体を表 4.2 に示す．型名:変数の四つの集合を Def 集合と呼ぶ．

仮引数定義は，伝播する仮引数の値が何番目の物かを示す定義で，各仮引数はメソッド解析の初めに自身の位置インデックスを持つ．具体的には，メソッド foo(A v1,A v2,A v3) を解析するときは，v1 の *Defpara* は {1}，同様に v2，v3 の *Defpara* は {2}，{3} を割り当てる．この割り当てを行う補助メソッドを DefparaInit() と定める．

変数名	説明
型名:メソッド	
CFG	ノードの集合
Para	仮引数の集合
ParaIndex	要約情報．デリファレンスされる仮引数番号の集合
returnValue	要約情報．返り値を表す値
型名:ノード	
instType	表 4.1 の部分言語のいずれかを格納
V	変数の集合
Pred	先行ノードの集合
型名:変数	
Defnull	ノードを要素とする null 定義の集合
Defobs	ノードを要素とする不明瞭定義の集合
Defnonnull	ノードを要素とする非 null 定義の集合
Defpara	整数値を要素とする仮引数定義の集合

表 4.2: 構造体の一覧表

4.2.2 解析のアルゴリズム

解析手法を擬似コードで説明する．

メソッド名:**methodAnalyze** 仮引数:*method*

```
1: DefparaInit(); //仮引数の各変数に仮引数定義を割り当てる
2: for 不動点に達していない場合 do
3:   for method.CFG 内のノード n 毎に do
4:     for n.V 内の全ての変数 v 毎に do
5:       変数 v の四つの Def 集合をそれぞれ先攻ノード n.Pred のすべてのノードの対応
         する変数の集合に対して和集合を求める (例:n.v.nullDef = n.pred1.nullDef ∪
         n.pred2.nullDef ∪ ...)
6:     end for
7:     if n.instType が InvokeAssign もしくは Invoke(innermethod(v1,v2,...vn)) then
8:       if innermethod.isnotAnalyzeStart()//呼び出し先が解析していなければ then
9:         methodAnalyze(innermethod);//呼び出し先を先に再帰的に解析する
10:      end if
11:      if Defnull が非空な v1,v2,...vn のインデックスが innermethod.ParaIndex に含まれていたら then
12:        警告を出力
13:      end if
```

14: **if** *n.instType* が InvokeAssign($v = innermethod(v1,v2,...vn)$) **then**
15: *AssignType* ← *innermethod.returnValue* //18 行目で使われる
16: **end if**
17: **end if**
18: **if** *n.instType* が NullAssign，ObsAssign，NonnullAssign もしくは *AssignType* に値がある場合 **then**
19: 該当する Def 集合を $\{n\}$ にし，他の三つの Def 集合は $\{\}$ にする
20: **else if** *n.instType* が VarAssign($v = w$) **then**
21: v の四つの Def 集合を右辺 w の四つの Def 集合と同じにする
22: **else if** *n.instType* が Dereference(v.dereference) **then**
23: **if** *v.Defnull* が非空 **then**
24: 警告を出力
25: **end if**
26: *method.ParaIndex* ← *method.ParaIndex* ∪ *v.DefPara*
27: **else if** *n.instType* が Return(return v) **then**
28: *v.Defnull* と *v.Defobs* によって，*method.returnValue* を決める
29: **end if**
30: **end for**
31: **end for**

4.2.3 解析の例

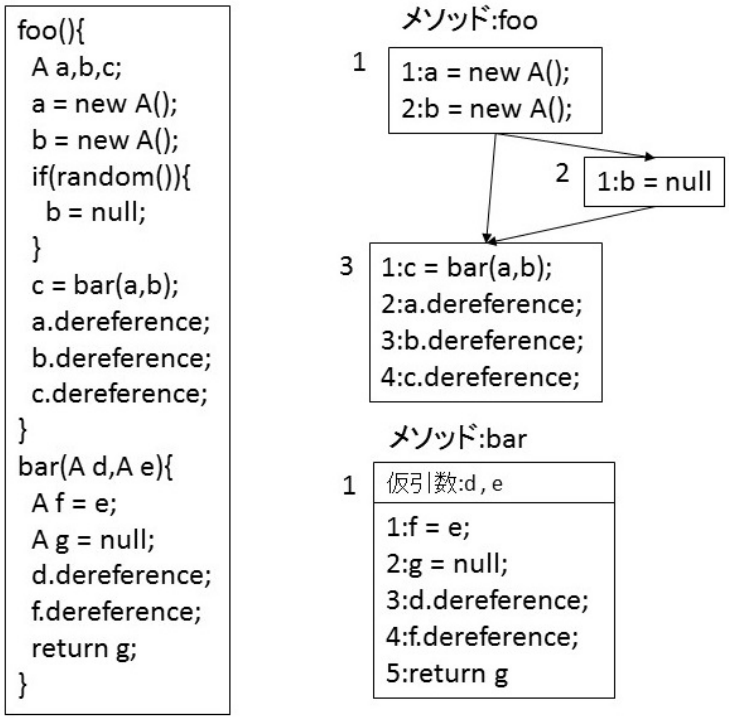

図 4.2: 解析の例のコードとグラフ

図 4.2 の左のコードをコントロールフローグラフとして表現したものが右のグラフである．

このグラフは簡略化のため，分岐のない命令のまとまりを大きな一つの節として表現しているが，解析で用いるノードは各命令単位ごとである．表 4.3，4.4 はメソッド foo，bar の各ノードの解析後の変数の Def 集合の状態等を示したものである．あるノードの解析時点で関係のない変数と集合は省略している．また，それらの集合はすべて空集合である．

4.2.2 のアルゴリズムの仕組みを図 4.2，表 4.3，4.4 を用いて解説する．

1 行目は初期化処理で，メソッドの仮引数がある場合に各仮引数の変数に対して自身の仮引数の位置インデックスを Defpara の要素にするメソッドである．具体的には表 4.4 の bar:初期化処理後である．

2-31 行目はすべてのノードのすべての変数の Def 集合が不動点に達するまで繰り返す．

4-6 行目は全ての変数の四つの Def 集合に対して，全ての先行ノードのそれらの集合との和集合を取る．例えば，foo:3-1 は，先行ノードが foo:1-2，2-1 であり，foo:3-1 の変数 b の Defnull は foo:1-2 と foo:2-1 の変数 b の Defnull の和集合を取ったものである．

7-17 行目は同じクラス内のメソッド呼び出しがある場合で，8-10 行目はその呼び出し先の解析が開始していなければ，呼び出し先のメソッドを再帰的に解析する．11-12 行目は呼び出し先メソッドの要約情報 *innermethod.ParaIndex* に，実引数の変数の中で Defnull が非空の変数がある場合，その変数の位置インデックスが *innermethod.ParaIndex* に含まれていると警告を出す．例えば，foo:3-1 を解析するとき，まず呼び出し先のメソッド bar を解析する．その後得られた bar の要約情報 ParaIndex は {1,2} で，これは呼び出し元の実引数の一番目と二番目の変数の Defnull が非空なら警告，と意味する．よって，foo:3-1 では，二番目の変数 b の nullDef が非空なので，警告を出す．14-15 行目は返り値を代入する場合で，解析したメソッドの要約情報 (*innermethod.returnValue*) の値に応じて 18-19 行目の処理をする．例えば，foo:3-1 では *innermethod.returnValue* は null である．よって，18-19 行目で変数 c の Defnull をその場所に，それ以外の Def 集合を空にする．

18-19 行目は値を代入する場合で，例えば foo:2-1 は b = null で null を代入するため，Defnull にその代入箇所があったノードを集合の値 {2-1} とし，他の三つの Def 集合は空集合にする．

20-21 行目は右辺に変数が来る場合で，左辺の四つの Def 集合を右辺の四つの Def 集合と同じにする．

22-26 行目は変数がデリファレンスされる場合で，この時にその変数が null とみなせると警告を出す．

23-25 行目はデリファレンスされる変数の Defnull が非空なら警告を出す．null 例外の発生箇所はこのデリファレンスがあったノードで，その原因となる null の代入箇所が Defnull の要素となる．例えば，foo:3-2，3-3，3-4 では各変数がデリファレンスされ，変数 b と c の Defnull が非空なので，foo:3-3，3-4 で警告を出す．

26 行目は仮引数の値がデリファレンスされる場合に，その仮引数の位置インデックスをメソッドの要約情報 *method.ParaIndex* に追加する．例えば，bar:1-4 では，変数 f は仮引数の位置インデックスが 2(変数 e) の値が伝播していて，それをデリファレンスしているので，ParaIndex に 2 を追加する．

27-29 行目は return 命令で，Defnull が非空の場合は null，Defnull が空で Defobs が非空の場合は obs，それ以外は非 null という値をメソッドの要約情報 *method.returnValue* に格納する．例えば，bar:1-5 では，Defnull が非空な変数 g を返しているので，returnValue を null にする．

変数名	Defnull	Defnonnull
foo:1-1		
a		1-1
b		
foo:1-2		
a		1-1
b		1-2
foo:2-1		
a		
b	2-1	
foo:3-1		
a		1-1
b	2-1	1-2
c	3-1	
foo:3-2〜3-4		
a		1-1
b	2-1	1-2
c	3-1	

表 4.3: foo の各ノード解析後の状態

変数名	Defnull	Defpara
bar:初期化処理後		
d		1
e		2
bar:1-1		
d		1
e		2
f		2
bar:1-2		
d		1
e		2
f		2
g	1-2	
bar:1-3		
d		1
e		2
f		2
g	1-2	
ParaIndex:{1}		
bar:1-4		
d		1
e		2
f		2
g	1-2	
ParaIndex:{1,2}		
bar:1-5		
d		1
e		2
f		2
g	1-2	
ParaIndex:{1,2}，returnValue:null		

表 4.4: bar の各ノード解析後の状態

4.3 配列

配列の解析は初期化方法を二種類に分け解析をする．ここでの初期化とは非 null の値が割り当てられることである．

解析に用いる配列の状態を表す構造体は，表 4.5 である．

変数名	説明
num	要素数．-1 なら不定値を表す
InitIndex	初期化済み要素番号の集合
allInitializedFlag	真偽値．すべての要素が初期化されたかどうか

表 4.5: 配列の状態を表す構造体

4.3.1 配列要素を一つずつ初期化する場合

配列要素を一つずつ初期化する例は

```
A a[] = new A[3];
a[0] = new A();
a[2] = new A();
```

のような配列の要素番号を指定し非 null に割り当てるような場合である．

配列の各要素が初期化されたかを解析するために，初期化済みの要素を表す集合 *InitIndex* を用いる．*num* と *InitIndex* の大きさが同じ場合は *allInitializedFlag* を true にする．上記の例では，配列 a の状態を表す構造体は，$a.num = 3$，$a.InitIndex = \{0,2\}$，*a.allInitializedFlag* = false である．

経路の合流点では集合の共通部分を求める．つまり，集合にある要素番号の要素は，すべての経路で初期化されていることを示す．

また，初期化子を用いた初期化 (例:`A a[] = {new A(), new A()}`) もバイトコード上では，各要素を初期化するのと同じことをしているので解析出来る．

4.3.2 すべての配列要素を一度に初期化する場合

すべての配列要素が初期化されたかどうかを真偽値 *allInitializedFlag* で表す．ループを用いてすべての配列要素を初期化する場合があり，これを解析するには，整数型変数を扱う必要がある．整数型変数は図 4.3 にある四つの状態間を遷移する．すべての変数は初期状態から始まる．全要素になった変数がループ内で配列の添字に使われ初期化されると 0 から比較された値までを初期化するとみなす．比較される値は定数値と配列の長さを表す.length のみ対応している．

ループで配列のすべての要素を初期化する例

```
1:A a = new A[10];
2:for(int i = 0 ; i < 10 ; i++){
3:  a[i] = new A();  }
```

を用いてオートマトンの遷移を説明する．整数型変数 i は”初期状態”から始まり，i = 0 で”値が 0”，i < 10 で”比較”，i++で”全要素”となる．その変数 i が 3 行目で配列の初期化に使われている．よって，その配列は 0-9 までの要素，つまり全要素が初期化されたとみなし，配列 a の状態を表す構造体は，$a.num = 10$，$a.InitIndex = \{\}$，*a.allInitializedFlag* = true である．

拡張 for 文で初期化する場合でも，上記の条件に該当するバイトコードが生成されるので，解析できる．

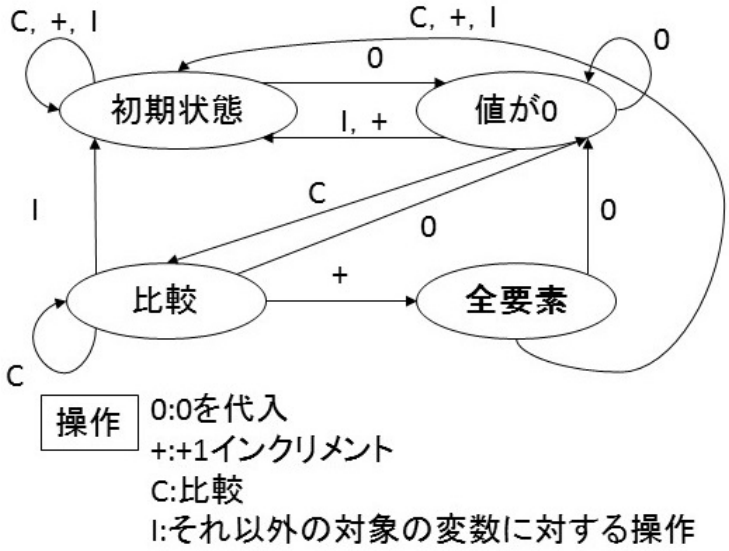

図 4.3: 整数型変数のオートマトン

また，ループ以外によく使われる配列の全要素初期化するパターンも対応する．配列初期化に用いられる toArray メソッド (例:C.toArray(array))，API の Arrays.fill メソッド (例:Arrays.fill(array, new A())) による初期化がある場合もすべての要素が初期化されたとみなす．

経路の合流点では *allInitializedFlag* の論理積を求める．つまり，*allInitializedFlag* が真であるというのは，その地点においてその配列は，すべての経路ですべての要素が初期化されたことを示す．

配列の初期化をこのように二種類に分けて解析する理由は，すべての場合において集合を用いて解析すると，配列の要素数が大きくなってしまった場合に計算量が増えるからである．

4.3.3 配列要素の参照

配列 (*array*) の要素を参照するとき，配列の添字とその配列の状態を表す構造体によって値を決める．

添字が定数の場合

その定数値が *array.InitIndex* に含まれる場合，もしくは *array.allInitializedFlag* が真のときは非 null とみなす．それ以外の場合は null とみなす．

添字が定数以外の場合

array.allInitializedFlag が真のときは非 null とみなす．それ以外の場合は null とみなす．

4.3.4 エイリアスと複製

エイリアスは一つの配列インスタンスを複数の変数でアクセスできることを指す．本研究は配列インスタンスのエイリアスに対応し，配列の変数ごとではなく，インスタンスごとに解析している．

また，配列の複製にも対応している．複製を行うメソッドの clone メソッドと System.arraycopy メソッドに対応し，それらにより配列インスタンスが複製される場合は，解析上でも配列の状態のインスタンスを複製している

4.3.5 配列と参照型変数

本研究は配列と参照型変数の解析を同時に行っているので，配列の要素を参照型変数へ代入，参照型変数を配列の要素に代入する場合も解析できる．

4.4 解析の工夫

誤検出を減らしたり，結果をわかりやすく示す工夫を説明する．

4.4.1 null 判定

参照型変数が if 文で null と比較があった場合，それを解析の結果に利用する．

・非 null と比較する場合

```
if(a != null){
  a.use();
}
```

if 文で変数 a は非 null と比較しているため，そのブロック内の a.use() は必ず a が非 null でデリファレンスされる．よって，変数 a の値が何であろうともこのデリファレンスでは警告しない．

・null と比較する場合

```
if(a == null){
  a = new A();
}
a.use();
```

変数 a が null の場合はブロック内で非 null になる．a が非 null の場合はブロックの終わりで非 null である．条件式を考慮しない場合は if ブロックの終わりで，a が null の場合は，null になる経路があるとみなしてしまうが，考慮することで if ブロックの終わりでは，変数 a が必ず非 null であると判断できる．

4.4.2 警告の重複を防ぐ

警告を出す変数が，値が更新されずに多くの場所でデリファレンスされていると，数だけ無駄に多くなってしまい，非常に見づらい．もし，それらのデリファレンスで実際にエラーが発生する場合，一番初めのデリファレンスで null 例外が発生し，動作が動作が停止してしまうため，それ以降のデリファレンスには到達しない．よって，初めのデリファレンスで警告を出力すれば，他は不要であることが多い．

本研究では，同じ null 定義の集合 (値) を持つ変数によって警告を出す場合，一番初めに起こるデリファレンスのみ警告する．

4.4.3 経路解析

本研究の参照型変数の null 解析には，null 代入の箇所の集合の伝播を求めている．その情報を用いて容易に経路を解析することができる．null 定義の集合が非空な変数がデリファレンスされるとき，null 定義の要素が示す場所との間の経路を求める．

経路の求め方は，null 集合が空でない変数のデリファレンスがある箇所から後方解析する．各 null 定義ごとにその null 定義が含まれている先行ノードを深さ優先探索する．その null 定義が含まれているノードが伝播する経路に相当する．

4.5 詳細な情報

null 定義と不明瞭定義と非 null 定義と仮引数定義を用いて，すべての経路で null になるかどうかを解析する．各変数の定義はこれら四つのうちのいずれかに該当するため，null 定義が非空で，他の三つの定義が空なら，すべての経路で null になるとみなし，警告時に表示を変えてより危険であると注意をうながす．

4.6 実装への工夫

バイトコードで解析するための工夫と高速化の工夫を説明する．

4.6.1 バイトコード解析

バイトコードで解析するために，表 4.1 の部分言語をバイトコードと対応させる必要がある．例えば null を代入する $v = \text{null};$ はバイトコードで表すと，

aconst_null

astore_1

となり，スタックに null 値を積んでそれを変数にストアするようになっている．よって，スタックの状態を解析する必要がある．また，デリファレンスするバイトコード命令はデリファレンスする箇所のスタックが null とみなせる場合に警告する．

4.6.2 高速化

本研究では高速化するために，コントロールフローグラフのノードを依存関係にしたがって順序付けをする．これを行うためにトポロジカルソートを用いた．そうすることにより，あるノードを解析するときは，先行ノードの解析が完了していることになり，計算の無駄が減る．

また，不動点を求めるのを，メソッドの全ノード単位ではなく，ループ単位で行うようにした．これにより，ループがない場合は不動点に達するまで計算をする必要がなくなる．

第5章　実装

本研究で提案した手法をツールとして実装した．クラスファイルの解析にはBCELを用いた．本ツールはコンソールからの実行に対応した．また，統合開発環境Eclipseのプラグインとして実装した (図5.1)．メニューバーのボタンをクリックすることで編集中のプロジェクトの全クラスファイルを解析する．その結果を図の左下のような文字で示すのに加え，警告箇所をエディタ上にマークする．警告された参照型変数をドラッグで指定し，パス解析というボタンを押すことで，null代入のある箇所からそこに至るまでの経路をマークする．

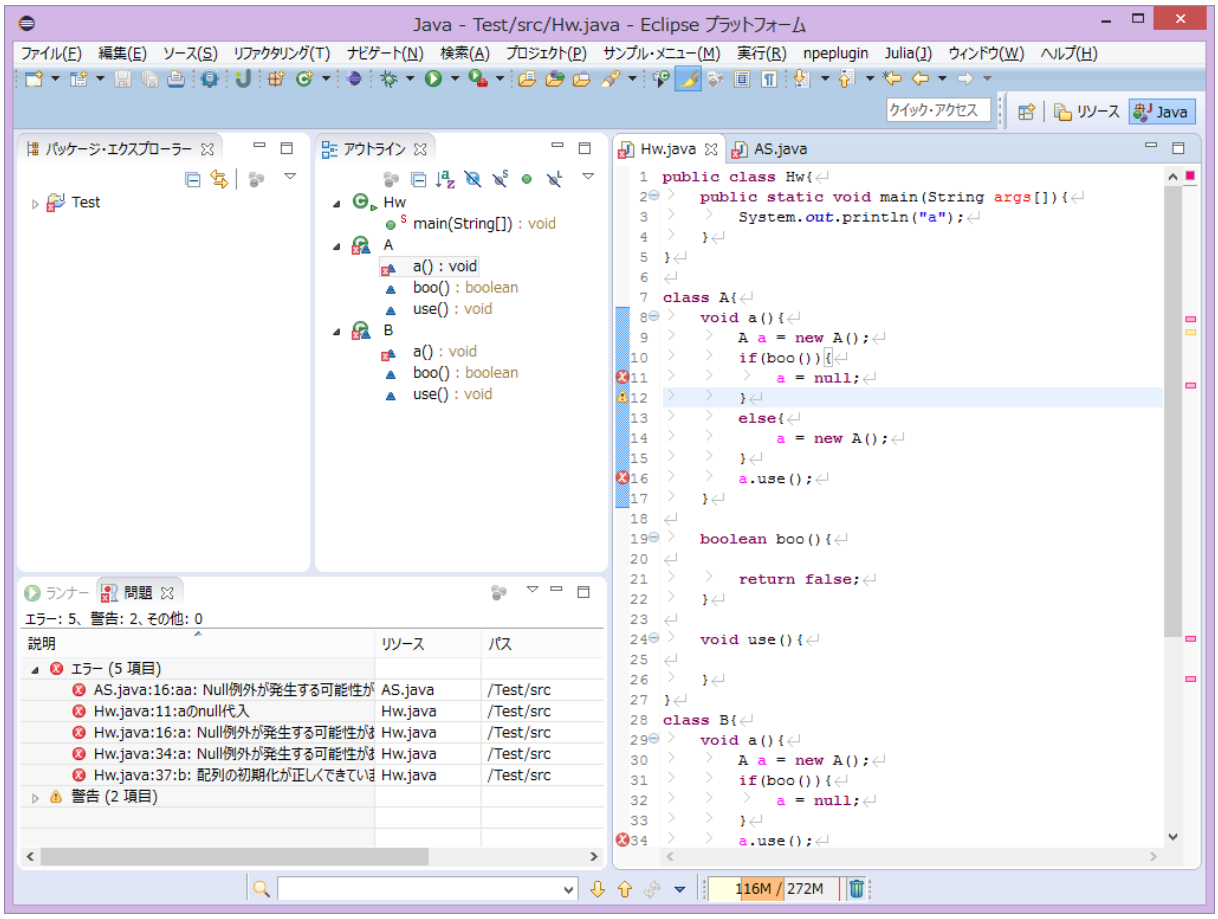

図 5.1: Eclipse での実行

第6章　評価

Intel Core i5-3470 64bit 3.20GH, 8GB RAM, Windows8.1, jdk 1.8 で本ツールを実行した．

6.1　実行例

意図的にバグを仕込んだコード (図 6.1) を解析し，そのときの出力を示す．

```
01:class Sample{
02:  void foo(){
03:    A[] a = {new A(),new A()} ,
04:    b = new A[2];
05:    for(int i = 0;i < 3;i++){
06:      b[i] = new A();}
07:    A x = null,y = null;
08:    if(bar()){
09:      x = new A();
10:      y = new A();}
11:    A[] c = {x,new A()};
12:    x.use();
13:    a[0].use();
14:    b[1].use();
15:    c[0].use();
16:    baz(x);
17:    if(y != null){
18:      y.use();
19:    }}
20:  void baz(A z){
21:    A x = z;
22:    x = z;
23:    x.use();
24:}}
```

図 6.1: 実行例

出力:
`Sample.java:12:x:` null 例外が発生する可能性があります
原因 `Sample.java:7`
`7-8 11-12`
`Sample.java:15:c:` 配列の初期化が正しくできていません
`Sample.java:16:` このメソッド呼び出しで null 例外が発生する可能性があります `Index:1` 呼び出し先:23

図 6.1 は上記の出力結果をわかりやすく色付けしている．変数 x は 7 行目で null が代入され，if ブロックを通らない場合は null になる．よって，12 行目のデリファレンスで警告を出している．原因は null 代入の箇所を示し，その下の数字は伝播する経路を示す．配列 c は 0 番目の要素は null になる可能性がある x なので，15 行目のデリファレンスで警告を出している．また，16 行目で baz が呼び出され，その仮引数の値は x に移った後にデリファレンスされている．よって，16 行目で警告を出し，呼び出し先の行も表示している．

それ以外は全て非 null をデリファレンスしている．よって，この例では本ツールは誤検出と検出漏れがなく，正しく動作していることが示せる．

6.2 実行結果

本ツールを，4 つのオープンソースソフトウェア (ant1.65，bcel5.2，jflex 1.61，cup0.1) と 5 つの本研究室で作られたソフトウェア (FlowAndValue，Seventh，NewTo，TougouPAD1.2，WindowsPrograming1.2) に対して実行し，その結果を表 6.1 に示す．行は解析対象のソースコードの行の合計である．有用な数というのは警告された場所を見て，コードを改善したほうがよいと判断した警告の数である．

ソフトウェア名	行	時間 (秒)	警告数	有用な数
ant1.65	176391	1.08	24	6
bcel5.2	48166	0.64	4	0
jflex 1.61	9125	0.34	1	0
cup0.1	1743	0.14	1	0
FlowAndValue	2487	0.20	0	0
Seventh	2965	0.25	0	0
NewTo	2171	0.22	0	0
TougouPAD1.2	2941	0.20	0	0
WindowsPrograming1.2	2442	0.19	1	1

表 6.1: 本ツールの実行結果

また，Findbugs3.0.1[2] でも同様のソフトウェアを解析する．null 例外に関係する物だけを警告数とする．結果を表 6.2 に示す．

ソフトウェア名	行	時間 (秒)	警告数	有用な数
ant1.65	176391	19.03	2	2
bcel5.2	48166	10.86	0	0
jflex 1.61	9125	3.21	0	0
cup0.1	1743	0.86	0	0
FlowAndValue	2487	2.80	1	0
Seventh	2965	1.35	0	0
NewTo	2171	1.48	0	0
TougouPAD1.2	2941	2.48	0	0
WindowsPrograming1.2	2442	2.53	4	4

表 6.2: Findbugs の実行結果

本ツールが警告した有用な例を二つ紹介する．いずれの Findbugs では警告しなかった例である．

例 1:ant1.65 Manifest.java

```
public String getKey() {
  if (name == null) {
    return null;
  }
  return name.toLowerCase();
}


154:String lhsKey = getKey();
155:String rhsKey = rhsAttribute.getKey();
156:if ((lhsKey == null && rhsKey != null)
157:                  || (lhsKey != null && rhsKey == null)
158:                  || !lhsKey.equals(rhsKey)) {
```

ant1.65 の Manifest.java の 154 行目で lhsKey は null が返される可能性のある getKey メソッドの返り値を参照している．158 行目で IhsKey と rhsKey が null のとき，null 例外が発生する．

例 2:WindowsPrograming1.2 Analysis.java

```
437:Data data=null;
    if(pattern==0){
     data = new Window(ss);
     if(w_connect==1){
     //並列に接続するタイプに変更
     if(Algorithm.parallel){
     data.connect_type=1;
     }
     if(old_data!=null && connect==1)
     data.step_parent=old_data.step_parent;
     }else if(w_connect==0){
     data.step_parent=data;
     }
    }else if(pattern==1){
     data = new Statements(ss);
     //w_connect_mode=0;//ウィンドウでなければ並列接続はしない
    }
    if(old_data!=null){
     if(connect == 0){
     old_data.content=data;
457: data.before=old_data;
```

pattern と w_connect はそれぞれ仮引数で，pattern と w_connect が 0 か 1 以外を取る場合に，457 行目で null 例外が発生する．

6.3 考察

表 6.1，6.2 より，本ツールは Findbugs と比べて解析時間は短く，警告数も多いという結果になった．Findbugs では解析出来ない手続き間からなる null 例外の警告も検出することができた．検出数自体も少なく，十分その部分のコードを確認できる数である．また，10 万行以上ある大規模なソフトウェアも動作が停止することなく解析することができた．しかし，誤検出 (警告数-有用な数) が非常に多くなってしまった．誤検出を減らすことが今後の課題である．

本ツールの誤検出はオープンソースソフトウェアを作るようなプログラミングに慣れた人が書く複雑なコードによって引き起こされる場合が多かった．特に配列の初期化時にループの上限値を定数や length 以外の数が使われていたり，メソッドの呼び出し先で初期化していたりした場合に誤検出した．複雑なコードが来ても正しく解析できるようにするのが今後の課題である．また，誤検出として検出された場合でも，その部分のコードは複雑になっていてわかりにくい場合であることが多いため，開発者はより簡潔でわかりやすいコードに変えたほうがよいと考える．

研究生が作る比較的単純なコードでは誤検出は起きなかった．よって，プログラムを自力で作るようになった人のような，単純なコードを書くが，null 例外を起こしやすい人が本ツールを使うことでよりよい効果が期待できると考える．

解析時間は非常に短いため，計算量が増えてでも検出能力を上げる方向性でツールを改善することがよいと考える．

第7章　結論と今後の課題

7.1　結論

本研究は，(1) 代入箇所の集合の伝播を計算することで，参照型変数における null 例外の発生箇所と，その原因となる null 代入の箇所とそれらの間の経路を解析，(2) 配列要素の番号の集合を用いることで，配列の各要素を null かどうかを解析，(3) 全要素を初期化するパターンを定め，配列の全要素が初期化されたか解析，(4) 引数と返り値から発生する null 例外を解析する手法を提案した．

それをツールの形として実装し，コンソールと統合開発環境 Eclipse の両方で解析することができる．実際にエラーとなる可能性が高い物を警告し，利用者が実際に目で見て確認できる量となっている．解析時間も一万行程度のプログラムなら数秒で終わるので，利用者にとって使いやすい物となっている．また，手続き間解析により，FindBug では見つけられないバグを見つけることができた．

7.2　今後の課題

バグであるものを警告する数を増やし，誤検出を減らすために，検出能力を向上させる必要がある．

7.2.1　フィールド変数

本研究では実装できなかったフィールド変数の解析の手法を説明する．

修飾子が private かそれ以外で解析の仕方を変える必要がある．修飾子が private の場合はすべてのコンストラクタとメソッド内で初期化されずにデリファレンスしていた場合に警告する．修飾子がそれ以外の場合は，同一メソッド内で一度 null が代入され，それがデリファレンスされる場合に警告する．

この手法をさらに発展させ，それをツールに実装するのが今後の課題である．

7.2.2　不明瞭な値

本研究ではフィールド変数やライブラリメソッドの返り値など解析できなくて null になる可能性がある物を不明瞭な値と定める．現状この値をデリファレンスするときに警告を出すようにすると，誤検出が多くなりツールとして使い物にならなくなる．よって，この値に該当するパターンを少しでも減らす工夫，もしくは有用な情報が得られるように改善する必要がある．そのためにはよりよい手続き間解析やフィールド変数の解析が必要である．

7.2.3 複雑なコードの解析

オープンソースソフトウェアを解析するときは誤検出が増えやすい．その誤検出の多くは工夫次第で消せるので，なぜ誤検出が起きるのかを理解し，出来る範囲で消していくことが今後の課題である．

特に，配列を手続き間で初期化しているのに対応したり，ループカウンタの上限値を柔軟にすると，誤検出が減らせる．

謝辞

本研究を進めるにあたり，色々とご指導を頂いた三重大学工学部情報工学科 講師の山田俊行先生，大野和彦先生に深く感謝致します．また，コンピュータソフトウェア研究室の皆様，研究活動における様々な場面で支えてくださった落合美子事務員に心からお礼申し上げます．

参考文献

[1] ORACLE. "The Java Language Specification Java SE 7 NullPointerException" `http://docs.oracle.com/javase/jp/7/api/java/lang/NullPointerException.html`

[2] W.Pugh and D.Hovemeyer, "FindBugs—Find Bugs in Java Programs", `http://findbugs.sourceforge.net/`.

[3] D.Hovemeyer, J.Spacco, and W.Pugh, "Evaluating and tuning a static analysis to find null pointer bugs", *Proc. PASTE '05 Proceedings of the 6th ACM SIGPLAN-SIGSOFT*, pp.13-19, 2005.

[4] D.Hovemeyer and W.Pugh, "Finding more null pointer bugs, but not noo many", PASTE '07 Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering, pp.9-14, 2007.

[5] D.Nikolic and F.Spoto, "Automaton-Based Array Initialization Analysis", *Proc. Language and Automata Theory and Applications*, pp. 420-432, Springer Berlin Heidelberg, 2012.

[6] R.Madhavan and R.Komondoor, "Null Dereference Verification via Overapproximated Weakest Pre-conditions Analysis", *Proc. Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, pp.1033-1052, 2011.

[7] M.Nanda and S.Sinha, "Accurate interprocedural null-dereference analysis for Java", IEEE 31st International Conference on, pp.133-143, IEEE, 2009.