情報処理学会論文誌 Vol.54 No.2 787–796 (Feb. 2013)

文字列解析の利用による動的クラスローディングに対応した静的データレース検出方法

魏 偉[1,a)] 吉浦 紀晃[1,b)]

受付日 2012年5月14日, 採録日 2012年11月2日

概要：マルチスレッドプログラムでは，スレッド間の排他制御においてミスが起こりやすい．排他制御が正しく行われないことに起因するエラーの１つとしてデータレースがある．データレース検出には動的手法と静的手法があり，動的手法はプログラム実行を通して得られた情報を解析することでデータレースを検出する．静的手法はプログラムのソースコードを調べることでデータレースを検出する．Java プログラムの静的データレース検出方法がすでに提案されているが，この静的データレース検出方法では動的クラスローディングが無視されている．そこで本論文では静的データレース検出方法における動的クラスローディングへの対策の方法を提案し，プログラムを実装する．さらに，開発したプログラムで Java プログラムのデータレースの検出を試行し，新たにデータレースを起こす箇所を発見した．これにより，本論文の検出プログラムの有効性を示した．

キーワード：データレース，静的解析，Java，動的クラスローディング

Static Data Race Detection for Java Programs with Dynamic Class Loading by Analysis of String Expressions

Wei Wei[1,a)] Noriaki Yoshiura[1,b)]

Received: May 14, 2012, Accepted: November 2, 2012

Abstract: Multi-thread programs are likely to have bugs in mutual exclusions between threads. Data race is one of the problems that occur from wrong mutual exclusions. There are two kinds of methods of data race detection: dynamic analysis detection and static analysis detection. Dynamic analysis detection is to detect data race by analyzing the results of program executions. Static analysis detection is to detect data race by analyzing source codes of programs. A method of static analysis detection of data race for Java programs has been proposed, but this method cannot deal with dynamic class loading. This paper proposes a method of static analysis detection of data race for dynamic class loading and implements this method. The experiment of this paper uses this method for Java programs that have dynamic class loading. This experiment shows that the proposed method can find possibilities of data race in Java programs more than previous data race detection methods.

Keywords: data race, static analysis, Java, dynamic class loading

1. はじめに

マルチスレッド・プログラムにおいてスレッド間の排他制御が正しく行われているかは重要であり，誤りがあると，プログラムの実行中に思わぬ障害が発生する．排他制御が正しく行われないことに起因するこの障害の主なものとしてデータレースがある．データレースとは，2 つのスレッドが同時に同じデータにアクセスして少なくとも 1 つのアクセスが書き込みであるときに発生するエラーである．この場合，アクセス順序によってはプログラム作成者の意図していない動作をプログラムが行うことがありうる．一方，データレースの検出は困難であることが知られている [14].

[1] 埼玉大学大学院理工学研究科
Department of Information and Computer Science, Saitama University, Saitama 338–8570, Japan
a) weiwei@fmx.ics.saitama-u.ac.jp
b) yoshiura@fmx.ics.saitama-u.ac.jp

データレース検出には動的手法と静的手法があり，動的手法はプログラム実行を通して得られる情報を解析することでデータレースを検出する [1], [16], [17], [18], [20]．静的手法はプログラムのソースコードを調べることでデータレースを検出する [6], [14]．動的手法はプログラムを実行することでデータレースを検出するため，プログラムを実行させるまでデータレースがあるかどうかが分からないという短所があるが，静的手法ではプログラムを実行せずに解析し網羅的にデータレースの検出を行いプログラム実行前にデータレースの有無が分かるため，プログラム実行における安全性を確保することができるという長所がある．一方，静的手法は網羅的に調べるため，データレース検出に大きな計算時間を必要とするという短所があるが，動的手法では計算時間が小さくてすむという長所がある．このように静的手法と動的手法にはそれぞれ長所と短所がある．

Naik らは，Java プログラムの静的データレース検出方法を提案している [14]．この検出方法は他の静的データレース検出方法よりも優れていることが実験により示されている [14]．具体的には，広く利用されている Java プログラムにおいて，多い場合には数百個程度の重大でこれまで知られていなかったデータレースをこの検出方法は検出し，さらに誤検出が少なかったことが実験により示されている [14]．この静的データレース検出方法では，5 つのステップを通してプログラムからデータレースを起こす可能性のあるメモリアクセスのペアを検出する．これらの 5 つのステップでは，最初のステップでデータレースの発生する可能性のあるメモリアクセスのペアの集合を作る．残りの 4 つのステップにおいて，この集合からデータレースを起こすことがないメモリアクセスのペアを取り除くことで，誤検出を極力なくし，データレースを起こす可能性のあるメモリアクセスのペアの集合を作り出す．

しかし，この検出方法では動的クラスローディングが無視されている．動的クラスローディングとは，Java プログラムを実行するときにクラスファイルが必要になったとき初めて Java 仮想マシンにクラスファイルがロードされる方式である．動的クラスローディングではネットワーク経由でクラスをロードできる．多くの Java プログラムに動的クラスローディングが使われているので，動的クラスローディングを無視すると重大なデータレースを検出できない恐れがある．

そこで，本論文では静的データレース検出方法における動的クラスローディングへの対策を提案する．Java では，実行時に動的クラスローディングを通してネットワーク経由でクラスをロードするが，クラスのロード元は URL で表される．静的手法によりデータレースを検出するためには，プログラム実行時のロード元となる URL を求める必要がある．本論文の提案する方法では，ロード元の URL を求め，この URL に基づいてクラスファイルをダウンロードする．次に，ダウンロードしたクラスファイルをデコンパイルしてソースコードを得る．得たソースコードをデータレースの検出対象であるソースコードと組み合わせてデータレースが発生するかどうか静的データレース検出方法 [14] で検出する．これによりデータレース検出漏れを防ぐことができる．そこで本論文では，動的クラスローディングへの対策の提案，提案した対策の実装，実装した対策によるデータレースの検出を実際に行い，提案した対策の評価を行う．

動的クラスローディングを含む Java プログラムに対するデータレースの検出には次のような問題がある．動的クラスローディングではプログラムの実行に応じてロードされるクラスファイルが異なる．プログラムによっては動的クラスローディングされるクラスファイルの組合せが多数になることもある．この場合，動的クラスローディングを考慮してデータレースの検出を行うと，クラスファイルの組合せの数だけ検出の手間が増えてしまい，検出自体の処理時間が大きくなる．また，動的クラスローディングの特徴は，プログラム実行時にロードされるクラスが決定されるため，無駄にクラスのロードを行う必要がないこと，クラス自体のメンテナンスを行う必要がないこと，つまり，Google MAP などのサービスを提供するサイトからの Web API 用のクラスファイルを動的にダウンロードすることで，つねに最新のクラスファイルを利用するといった利点がある．逆にこれがデータレースの検出を難しくする原因にもなっている．

本論文の構成は以下のとおりである．2 章で本論文で利用する静的データレース検出方法の説明を行う．3 章で Java 動的クラスローディングの説明を行い，4 章で動的クラスローディングに対応したデータレース検出方法を提案する．5 章で本論文で提案した検出方法の評価を行うための実験を行い．6 章で関連研究を述べ，7 章で本論文をまとめる．

2. 静的データレース検出方法

提案された静的データレース検出方法 [14] は，**図 1** に示すように 5 つのステップに分かれる．この 5 つのステップは，Original-pairs 計算，Reachable-pairs 計算，Aliasing-pairs 計算，Escaping-pairs 計算，そして Unlocked-pairs 計算である．この静的データレース検出方法には次の特徴がある [14]．

- 誤検出の確率はほかのデータレース検出方法より低い．
- 65 万行程度までの大規模プロジェクトでも使用に耐える．
- Java プログラムに使われている Synchronized メソッド，wait メソッド，notify メソッドに対応している．
- ライブラリ自体への対応はできていない．
- データレースを起こすプログラムの動作例を出力できる．

また，Web サーバである Apache の 1 つのコンポーネン

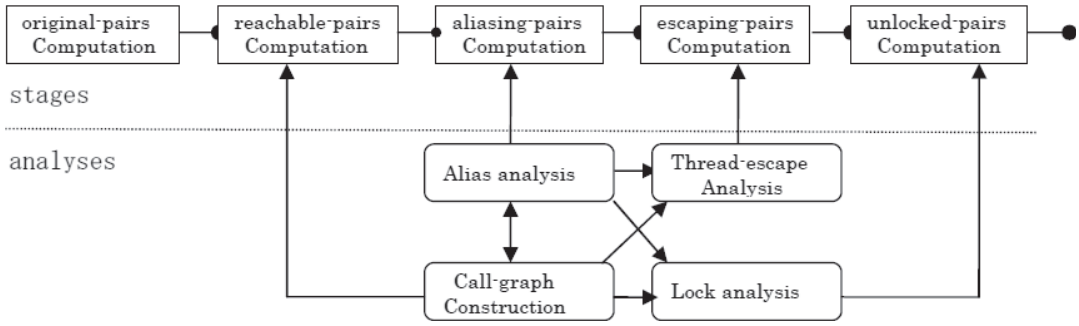

図 1 静的データレース検出方法の流れ
Fig. 1 Overview of static detection of data race.

トである Apache commons-pool に，この静的データレース検出方法を適用した事例がある [14]．この事例では，この検出方法によりデータレースの発生する箇所が 17 カ所指摘され，そのうち 5 カ所で実際にデータレースが発生することが確認された．この事例における指摘箇所とデータレースの発生箇所の個数は，誤検出の確率がほかのデータレース検出方法より低いことを示す 1 例になっている．

```
Main() {
 Object x,y;
 Thread1(x,y).start();
 .......
 Thread2(x,y).start();
 .......
}
```

図 2 Thread-escape 解析
Fig. 2 Thread-escape analysis.

2.1 静的データレース検出方法のアルゴリズム

本論文で利用する静的データレース検出アルゴリズム [14] は次の各ステップから構成される．

- Original-pairs 計算
  同時に同じ箇所へアクセスする可能性のあるメモリアクセスのペアをすべて列挙する．この Original-pairs 計算を通して得た結果を OriginalPairs とする．具体的には，インスタンス変数，static 変数，配列への読み込みと書き込みまたは書き込みが同時行われる可能性のあるメモリアクセスのペアを列挙する．この Orignal-pairs 計算の結果である OriginalPairs の中にはデータレースが発生しないメモリアクセスのペアが含まれている．以後のステップでは，データレースが発生しないメモリアクセスのペアを取り除く．
- Reachable-pairs 計算
  コールグラフを作成し，main メソッドから到達可能ではないメモリアクセスのペアを取り除く．コールグラフとはサブルーチンどうしの呼び出し関係を表現した有向グラフであり，Reachable-pairs 計算では初めにこのグラフを作成する [7]．これは，main メソッドから到達可能なスレッド間でしかデータレースが発生しないためである．Reachable-pairs 計算を通して得た結果を ReachablePairs とする．
- Aliasing-pairs 計算
  x, y をインスタンス, f を変数とする．ReachablePairs に (x.f, y.f) が含まれる場合，つまり，インスタンス x が変数 f にアクセスし，インスタンス y も変数 f にアクセスする場合，Java プログラムにおいて x=y という命令があるならこれら 2 つのメモリアクセスはデータレースを起こす可能性がある．Aliasing-pairs 計算では，エイリアス解析 [13] により引数の参照渡し・参照変数・ポインタを介した参照などで生じる異なる変数が同じオブジェクトを指すかを調べる．この解析により，メモリアクセス参照が同じメモリ領域を指す可能性のないペアを取り除く．Aliasing-pairs 計算を通して得た結果を AliasingPairs とする．
- Escaping-pairs 計算
  Thread-escape 解析により複数のスレッドに共有される可能性のないメモリ領域を参照しているメモリアクセスのペアを AliasingPairs から取り除く．Thread-escape 解析とは複数のスレッドに共有されるオブジェクトを見つけ出す解析である．Java プログラムにおいて複数のスレッドに共有される可能性のあるメモリ領域は，スレッドを起動するメソッドにおける引数と static フィールドである．
  たとえば，**図 2** のように x，y はスレッドを起動するメソッドにおける引数であり，複数のスレッドに共有されている可能性がある．Thread-escape 解析に加え，エイリアス解析により複数のスレッドに共有される可能性のあるすべてのメモリ領域を見つけ出す．最後に AliasingPairs に含まれるメモリアクセスのペアのうち，複数のスレッドに共有されている可能性のないメモリ領域を指しているメモリアクセスのペアを取り除く．この Escaping-pairs 計算を通して得た結果を EscapingPairs とする．
- Unlocked-pairs 計算
  Lock 解析により共通のロックを持っているメモリア

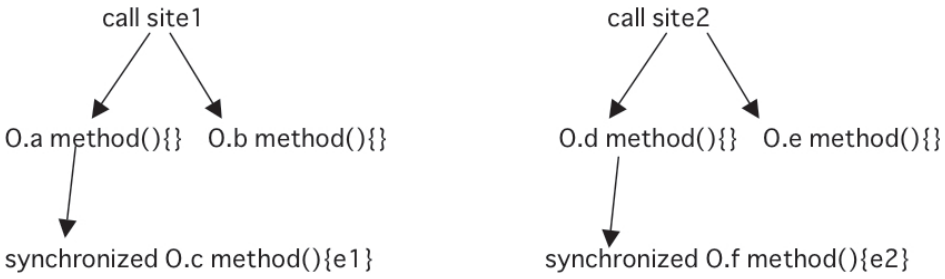

図 3 Unlock-pairs 計算
Fig. 3 Unlock-pairs computation.

クセスのペアを取り除く．Java ではロックを利用することで排他制御を実現でき，共通のロックを持っている場合，データレースを起こさない．

Lock 解析とはコールグラフ上でスレッドを起動するメソッドから，メモリアクセスを含むメソッドへのすべてのパスを検索してペアになるメモリアクセスに，コールグラフ上でどのパスを通っても共通のロックがあるかどうかを調べるものである．**図 3** において，call site1 と call site2 をスレッドを起動するメソッド，O をオブジェクト，e1 と e2 をメモリアクセスとする．call site1，O.a method()，synchronized O.c method() と call site2，O.d method()，synchronized O.f method() という 2 つのパスは共通のロックを持っているためデータレースを起こさない．このようなメモリアクセスのペアを EscapingPairs から取り除く．Unlocked-pairs 計算を通して得た結果を UnlockedPairs という．

静的データレース検出方法 [14] では，UnlockedPairs に含まれるメモリアクセスのペアをデータレースの可能性があるものとする．しかし，UnlockedPairs に含まれるものがすべてデータレースを起こすものであるとは限らず，メモリアクセスのペアがデータレースを起こすかの判断には人手が必要である．

2.2 静的データレース検出方法の性質と問題点

提案された静的データレース検出方法 [14] には以下に示す問題点がある．

(1) Aliasing-pairs 計算などで利用しているエイリアス解析は，must-alias 解析ではなく，may-alias 解析である．このためデータレース検出が正しく行われない場合がある．
(2) ライブラリなど単体では使われないプログラムにおけるデータレースを調べるためには，呼び出しメソッドの定義が必要である．呼び出されないメソッドの場合は main メソッドを補い，データレースの検出を行う．しかしすべての状況をシミュレーションできないのでデータレース検出漏れの可能性がある．
(3) 他の多くの静的なデータレース検出方法と同様に，initializer，constructor，finalizer において発生するデータレースを検出できない．
(4) Java のリフレクション機能や動的クラスローディングを無視しているため，これらの機能を利用したプログラムでのデータレースを検出できない．

本論文ではこれらの問題点の (4) の中でも動的クラスローディングへの対策を提案する．多くの Java プログラムは動的クラスローディングを利用している．特に分散システムにおいては，動的クラスローディングを通してサーバやインターネット上からクラスファイルをロードしてローカルにあるクラスファイルのように Java 仮想マシンにロードされる場合が多い．よって，動的クラスローディングを考慮した静的データレース検出方法が必要である．本研究では，動的クラスローディングに対応した静的データレース検出方法を提案する．

3. Java 動的クラスローディング

3.1 Java のクラスローダ

Java においてクラスファイルを実行するには，クラスファイルを Java 仮想マシンにロードする必要がある．クラスファイルをロードするコンポーネントをクラスローダと呼ぶ [10]．Java ではライブラリを JAR ファイルに格納し様々なオブジェクトを格納することができる．Java のクラスローダはライブラリを見つけロードし，ライブラリに含まれるクラスファイルをロードする．

図 4 のように Java のクラスローダは主に 3 つの種類に分けられる．3 つの種類はブートストラップクラスローダ，拡張クラスローダ，システムクラスローダである．ブートストラップクラスローダの役割は Java 標準のライブラリ（java.∗ パッケージ，javax.∗ パッケージ）をロードすることである．拡張クラスローダの役割は JRE の拡張ディレクトリ（⟨JRE_HOME⟩/lib/ext）内のクラスをロードすることである．システムクラスローダの役割はシステムの CLASSPATH 変数で指定されたディレクトリや JAR ファイル内からクラスをロードすることである．

システムクラスローダを拡張することによって，様々なタイプのクラスローダを実装することができる．拡張されたクラスローダではインターネットからクラスファイルをロードすることが可能である．

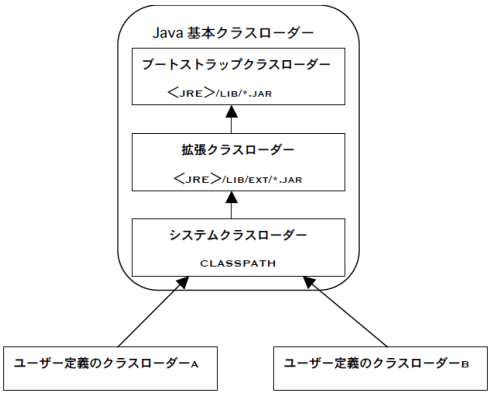

図 4 Java のクラスローダの構成
Fig. 4 Structure of Java class loaders.

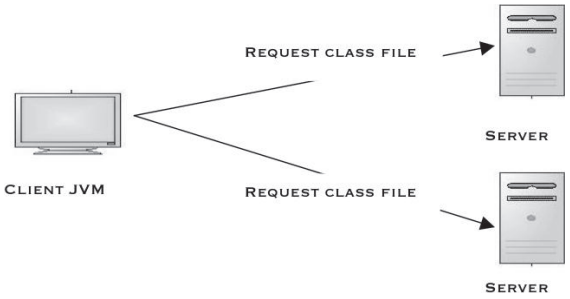

図 5 動的クラスローディング
Fig. 5 Dynamic class loading.

3.2 動的クラスローディング

Java プログラムが必要とするクラスファイルは，通常，プログラム実行中にそのクラスファイルの実行が必要になったときに初めて Java 仮想マシンにロードされ実行される．これを動的クラスローディングと呼ぶ．**図 5** のように Java プログラムを実行するときにユーザ定義のクラスローダを利用することによって Client の方にある Java 仮想マシンはインターネットを経由し，サーバからクラスファイルを読み込み，Java 動的クラスローディングを実現する．

ユーザ定義のクラスローダを作成するときに Java ではクラスファイルのロード元を URL で表す．URL は 3 つのアクセスをサポートしている．3 つのアクセスは，HTTP プロトコルと FTP プロトコルとローカルファイルシステムである．URL は，ユーザからの入力，コンフィグファイルやプロパティファイル，プログラム実行中の計算結果などに依存している．静的データレース検出方法 [14] は，プログラムのソースコードを解析することでデータレースを検出するため，プログラム実行中でなければ判明しないロード元 URL を求めることができず，動的クラスローディングを無視している．

4. 動的クラスローディングへの対策

4.1 変換処理

動的クラスローディングへの対策には，クラスファイルのロード元 URL の値を計算することが必要になる．URL は計算するには次の 3 つのパターンを考える必要がある．つまり，ユーザからの入力に依存しているパターン，変数間の演算に依存しているパターン，そしてコンフィグファイルやプロパティファイルに依存しているパターンである．提案する対策では URL の値が依存する変数などのデータフローグラフを作成し，URL がとりうる値をできるだけ多く求める．データフローグラフの作成を容易にするため，ソースコードの中の出現する繰返し文，例外処理，メソッドの呼び出し，配列に対して以下に説明する処理を行う．

```
while (Cond)
 Body;
```

⟹

```
if (Cond) {
  Body;
  if (Cond) {
    Body;
    if (Cond) {
    ......
    }
  }
}
```

図 6 while 文の変換
Fig. 6 Conversion of while sentence.

- 繰返し文の変換
  URL の値を計算するために，繰返し文の分析が必要ならば，繰返し文の削除を行う．繰返し文には主に 3 つの種類がある．for 文，while 文，do-while 文である．for 文については繰返しの回数が定数などにより決まっている場合には，繰返し文を削除し繰り返されるプログラムの部分を繰返しの回数分だけ実行するようにプログラムを修正する．繰返しの回数が変数により決まるため繰返しの回数を決められない場合は，繰返しの回数を適当に定め，繰り返されるプログラムの部分をこの回数分だけ実行するようにする．本論文で実際にプログラムを解析する場合には，繰返しの回数を 1〜5 回と定め各回数についてデータレースの検出を行う．while 文や do-while 文については，**図 6** にあるように while 文を削除し，条件判定文を利用してあらかじめ定めた回数繰り返すように変換する．繰返し文がネストしている場合には，外側の繰返し文から順次繰返し文を削除していく．
- 例外処理の変換
  Java ではプログラムの実行中に発生するエラーを例外として扱うことができる．例外処理においてよく使われているのが try-catch である．URL の値を計算するために try-catch の解析が必要ならば，try で記述されている部分を，条件判定文と catch の部分を追加することで try-catch の削除を行う．
- メソッドの呼び出しの変換
  URL の値を計算するためにメソッドの呼び出しが必要な場合，この部分に呼び出されたメソッドのソース

コードをコピーする．呼び出されたメソッドに返り値があるならば代入の形でのコピーを行う．データレースの検出対象となるソースコードは増えるが，メソッドの呼び出し処理の部分がなくなるため URL の値の計算を簡単化することができる．なお，メソッドの呼び出しが再帰的になっている場合もあるため，コピーの回数には上限を設けて行う．

- 配列の変換
  URL の値を計算するために配列の要素が必要である場合，インデックスが決まっている場合にはそのインデックスに対応したの配列の値を配列となる変数に置き換える．決まらない場合には，配列の各要素の値がとりうる値であると見なし，すべて列挙したものを解析する対象のソースコードとする．

4.2 データレース検出方法

動的クラスローディングを考慮したデータレース検出方法は，2 つのステップからなる．1 つはクラスファイルのロード元の URL を求めること，もう 1 つはクラスファイルをロードしてそのクラスファイルを合わせたソースコードに対して静的データレース検出を行うことである．

4.2.1 ロード元 URL の計算

動的クラスローディングを考慮したデータレース検出では，クラスファイルのロード元 URL を調べる必要がある．ソースコードを直接検索して URL が定数であれば，それを利用するが，定数ではない場合には，次のステップによりとりうる値を求める．なお，以下のステップは，プログラムに前述した変換処理を行ったものに対して適用する．

定数ではない場合は，URL の値を求めるために必要な変数の依存関係を調べデータフローグラフを作成する．**図 7** にその例を示す．この図のように，URL や変数の値を求めるのに必要なものが複数ある場合には &，とりうる値が複数個ある場合には，+ によりすべてをグラフ上に記載する．変数の依存関係は，ソースコードを後方解析することにより得る．ソースコードを解析する手順は以下のとおりである．URL を表す変数の代入文を検索する．代入文の右辺が定数であれば検索を停止する．代入文の右辺が定数でない限りこの検索を繰り返し，代入文の右辺が定数，ユーザの入力，コンフィグファイルやプロパティファイルのいずれかに依存している場合に終了する．

4.2.2 データレース検出

クラスファイルのロード元 URL が判明したものをすべてダウンロードする．クラスファイルは JAR ファイル形式であるので，デコンパイルし，Java のソースコードを求める．JAR ファイルから Java のソースコードへの変換は，定数どうしの演算など最適化が行われている部分以外は可能である．このクラスファイルのソースコードと元の Java プログラムのソースコードを組み合わせてデータレースの検出を行う．ロード元の URL が複数ある場合には，それぞれの URL からダウンロードしたクラスファイルごとにデータレース検出を行う．

4.3 実装

本論文の提案方法を Java により実装した．

このプログラムは，検出対象となるソースコードを受け取り，システムクラスローダやユーザ定義のクラスローダを確定させ，変換処理によりソースコードの中の繰返し文，例外処理，メソッドの呼び出し，配列の部分を変換する．次にクラスファイルのロード元 URL を求め，求めることができた URL からクラスファイルをダウンロードしデコンパイルしてソースコードと合わせデータレースの検出を行う．データレース検出部分については，提案された静的データレース検出方法 [14] を実装した Chord [4] を用いる．よって本論文での開発部分はそれ以外の部分となる．

Chord の中では Original-pairs 計算において，同時に同じ箇所へアクセスする可能性のあるメモリアクセスのペアをすべて列挙するために Soot ツール [21] を利用している．このツールは Java プログラムを解析するツールであり，クラスファイルの解析やヌルポインタの解析や制御フローやデータフローの解析などが実現できる．

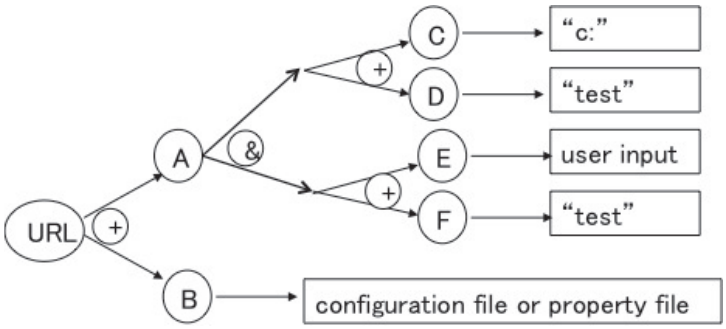

図 7 URL 計算のための変数の依存関係
Fig. 7 Dependency relation of variables for URL.

5. 実験

5.1 実験方法

本論文で開発したプログラムを次の環境で動作させ，プログラムの有効性を調べた．動作環境は，OS は Windows XP version 2002，CPU は AMD Athlon^{TM} 64 Dual Core Processor 2.0 GHz，メモリは 2 GB である．

動的クラスローディングを利用している 3 つのソフトウェアを対象として実験を行う．fileStream はファイル転送を行うためのソフトウェアである．動的クラスローディングにより転送のファイルのタイプやサイズをチェックするためのクラスファイルをロードする．weatherService は日付と地方を入力して，日付からの 1 週間の天気を調べるソフトウェアである．動的クラスローディングにより，調べたあとの 1 週間の天気を表示するためのクラスファイルをダウンロードする．examSystem は受験ソフトウェアである．ID と Password を入力し試験の問題を選択し回答したあとで正解かどうかチェックする機能がある．動的クラスローディングにより，チェックしたあとの結果によってどんな科目が苦手で，どんな科目が得意であるか，どんな科目をより勉強したほうがいいかを推薦するためのクラスファイル，そして各科目の成績を各科目の成績の平均値と比較する折れ線グラフを作って表示するためのクラスファイルをダウンロードする．

これらのソフトウェアを選んだ理由は，ソフトウェアのソースコードを公開していること，動的クラスローディングが利用されていることである．これらのソフトウェアの特徴を**表 1** に示す．

5.2 実験結果

3 つのソフトウェアを対象にして実験を行う．この実験では，従来の静的データレース検出方法 [14] によるデータレース検出と本論文で提案した方法による静的データレース検出をそれぞれ適用しその結果を比較した．**表 2** が従来の方法の結果，**表 3** が本論文で提案した方法の結果である．各表では，検出までの時間，データレース検出方法の各計算ステップで得られる OriginalPairs，ReachablePairs，AliasingPairs，EscapingPairs，UnlockedPairs の要素数，実際にデータレースが起きるメモリアクセスのペア（DataracePairs）の数が掲載されている．本論文の利用している検出方法では，UnlockedPairs に含まれていても必ずしもデータレースを発生させるメモリアクセスのペアとはいえない．そのため，UnlockedPairs に含まれるメモリアクセスのペアの中で確実にデータレースが発生すると判断できるペアを人手により調べ，その結果を DataracePairs として示している．これら 2 つの表を比較すると，本論文により提案された方法により新たにデータレースを生じるメモリアクセスのペアが見つかった．特に，weatherService と examSystem の 2 つで新たに見つかったデータレースを発生させるメモリアクセスのペアは動的クラスローディングによりロードされたクラスファイルに関連しており，本論文で提案した方法により初めて見つけることができたものである．

このデータレースは動的クラスローディングによりロードされたクラスファイル内の static フィールドが複数のスレッドで同時にアクセスされるために生じる．このデータレースはクラスファイルが複数のスレッドで同時に利用されるだけでは起きず，その利用方法にも依存する．よって，その利用方法によってはデータレースを起こさないため，クラスファイル単体で検証するだけでは発見できない可能性がある．しかし，この動的クラスローディングされるクラスファイルが必ずしも動的にローディングされる必要がない場合，プログラムの安全性を考えると，動的クラスローディングを利用せずに利用するクラスファイルをあらかじめ用意することでデータレースの検出を容易にするほうがよいと思われる．

5.3 考察

本論文で提案した静的データレース検出方法では，新たに

表 1 対象となるソフトウェア
Table 1 Target software.

ソフトウェア	クラスの個数	動的クラスローディングされるクラスファイル数	行数
fileStreaming	5	1	638
weatherService	18	2	1,598
examSystem	35	2	6,653

表 2 動的クラスローディングを無視した場合
Table 2 Without dynamic class loading.

Project	Time	Original Pairs	Reachable Pairs	Aliasing Pairs	Escaping Pairs	Unlocked Pairs	Datarace Pairs
file Stream	5 s	456	28	15	15	4	0
weather Service	19 s	1,762	117	87	74	19	1
exam System	52 s	3,725	220	106	104	8	1

表 3 動的クラスローディングを考慮した場合
Table 3 With dynamic class loading.

Project	Time	Original Pairs	Reachable Pairs	Aliasing Pairs	Escaping Pairs	Unlocked Pairs	Datarace Pairs
file Stream	17 s	526	46	32	29	9	0
Weather Service	47 s	2,062	185	137	122	23	3
Exam System	157 s	4,465	323	253	247	12	2

データレースを起こすメモリアクセスのペアを発見することができた．一方，検出時間も増加している．examSystem では，動的クラスローディングを無視した場合の 3 倍増加している．weatherService や examSystem での増加した処理時間のうち約 80%は URL の値の計算などに利用されており，ソースコードの規模が大きくなるとデータレースの検出自体と同様に URL の値の計算に時間がかかると推測される．プログラムの構造が複雑になるためプログラムの変換に時間がかかることや変数の値を求めるためのデータフローグラフが大きくなることから URL の値の計算に時間がかかると推測される．よって，URL の値に計算の効率化やプログラム自体の改良が必要になる．また，動的クラスローディングによりロードされるクラスファイルがロード元 URL の値により複数種類あり，動的クラスローディングがプログラム中で複数箇所で利用されている場合，実行時に利用される可能性のあるクラスファイルの組合せが複数になることもある．この場合，各組合せに対してプログラムのソースコードと合わせてデータレース検出を行う必要があり，すべての組合せに対してデータレース検出を行うと時間がかかる．

各 Pairs の要素数であるが，weatherService や examSystem では，最終結果である UnlockedPairs がそれぞれ 19 から 23，8 から 12 と増加しているが，データレースを起こすメモリアクセスのペアがそれぞれ 2 個，1 個と発見されており，UnlockedPairs の増加に比べると効率良く検出されているといえる．

本論文の検出法方で新規にデータレースが見つかった 2 つのソフトウェアについて，weatherService では，OriginalPairs が 1.17 倍，ReachablePairs が 1.58 倍，AliasingPairs が 1.54 倍，EscapingPairs が 1.65 倍，UnlockedPairs が 1.2 倍に増えている．examSystem では，OriginalPairs が 1.2 倍，ReachablePairs が 1.46 倍，AliasingPairs が 2.39 倍，EscapingPairs が 2.38 倍，UnlockedPairs が 1.5 倍に増えている．fileStream についても同じように各 Pairs の要素が増えている．最終結果である UnlockPairs に比べ，AliasingPairs と EscapingPairs が大きくなっており，これらの Paris の構成方法を検討する必要がある．また，AliasingPairs と EscapingPairs の要素数にそれほど大きな変化がないことから，Escaping-pairs 計算を省くことにより時間の短縮化の可能性がある．

本論文では，実験対象として 3 つの Java プログラムを利用した．これらのプログラムでは動的クラスローディングのロード元となる URL が変数であり，URL を表す文字列自体の代入といった簡単な操作によりその変数の値が判明するものではない．このことが実験対象のプログラムとして選んだ理由である．

本来，文献 [14] において実験対象としている Java プログラムを本論文の実験対象とすべきである．しかし，データレース検出方法により検出されたメモリアクセスのペアが，実際にデータレースを起こすかを確認する必要があり，この確認は人手によるため時間がかかる．また，ロード元となる URL の値を求めることが簡単ではないことが必要であったため，本論文では文献 [14] にあるプログラムを実験の対象とはせずに，ロード元の URL の値を求めることが簡単ではない別の Java プログラムを本論文の実験対象とした．一方，本論文の実験対象である Java プログラム weatherService はクラス数が 18，行数が 1,598，examSystem はクラス数が 35，行数が 6,653 であり，examSystem は文献 [14] にある実験 JDK 1.1 java.util.Vector や JDK 1.1 java.util.Hashtable の 3 倍弱の規模であり，本論文の実験においても，提案している方法の有効性は示せていると考えられる．

Java プログラムの規模が大きい場合，問題となるのはデータレース検出方法で導出されたメモリアクセスのペアが本当にデータレースを起こしているかを確認することである．これに対する解決策の 1 つとして，モデル検査手法を用いてメモリアクセスのペアがデータレースを起こすのかを調べることが考えられる．

6. 関連研究

本論文では，動的クラスローディングを行うための，クラスファイルのロード元が変数で表されている場合，この変数に対してデータフローグラムを作成し，変数の値となりうる文字列を可能な限り求めている．よって，本論文は，データフロー解析と密接な関係がある．データフロー解析の研究は古く [12]，特に，本論文は Java の文字列解析の研

究と関連がある．Java String Analyzer [5], [8] では，Java プログラムから文字列を表す変数がとりうる値を厳密に導き出し，そのとりうる値を文脈自由文法により表現する．一方，本論文では，厳密に変数の値を求めることは行っていない．

動的クラスローディングを行うロード元は，複雑な文字列であることは少なく，正規表現における繰返しなどの演算により表現されることはほとんどない．そこで，本論文では，Java String Analyzer のようにプログラムの構造から厳密に変数の表現を求めることは行わず，プログラム中の繰返し文などを変換することで，変数の値となりうる文字列を求める方法を簡単にしている．よって，本論文の方法では文字列解析に必要な時間やメモリ量は多くはなく，文字列を求めるためのメモリ量は 1M バイト以下である．一方，Java String Analyzer の場合，4,000 行弱のあるプログラムの処理のために 107M バイトのメモリを必要とし [5]，必要なメモリが本論文の方法よりも多い．

Sawin らの方法 [19] では Java String Analyzer を応用することで動的クラスローディングのロード元を表す変数の値を求める方法を提案している．具体的には，Java String Analyzer に実行時の環境変数を考慮させることで，変数の値となる候補をより多く見つけ出すことができる．本論文の方法では文献 [19] と同様に環境変数を考慮しているが，一方で Java String Analyzer ほど厳密にプログラムの構造から変数の値を求めない代わりに変数の値となりうる文字列を求める方法を簡単にしている．

一方，文献 [14] では，動的クラスローディングのほかにもリフレクションを無視してデータレースの検出を静的手法によって行っている．文献 [11] では，Java のプログラムにおけるリフレクションを考慮してコールグラフを作成する方法を提案している．このコールグラフを文献 [14] の静的データレース検出方法に応用することで，リフレクションを考慮した静的データレース検出手法ができる可能性がある．また，モデル検査器の 1 種である JavaPathFinder を利用したデータレース検出方法 [9] がある．この方法では検出されたメモリアクセスのペアはデータレースを起こすことは保証されるが，文献 [14] のようにデータレースを起こすメモリアクセスのペアを多く検出することはできていない．他の研究としては動的手法での Java のデータレース検出方法 [3]，Java 以外の言語でのデータレースの検出の研究 [2] などがある，しかし，本論文のような動的クラスローディングに関連したデータレースの静的検出手法の研究は見受けられない．

7. おわりに

本論文では静的データレース検出における動的クラスローディングへの対策を提案し，この提案方法を実装し，実験により本論文で提案した方法により新たにデータレースを起こすメモリアクセスのペアを見つけることができた．

今後の課題としては，URL の値の計算の効率化があげられる．また，UnlockedPairs にはデータレースを起こさないメモリアクセスのペアが含まれているが，そこからさらにデータレースを起こさないメモリアクセスのペアを排除する方法の提案などがあげられる．

参考文献

[1] Agarwal, R., Sasturkar, A., Wang, L. and Stoller, S.: Optimized run-time race detection and atomicity checking using partial discovered types, *Proc. 20th IEEE/ACM International Conference on Automated Software Engineering* (*ASE'05*), pp.233–242 (2005).

[2] Blanc, N. and Kroening, D.: Race analysis for SystemC using model checking, *ACM Trans. Design Automation of Electronic Systems*, Vol.15, No.3, pp.21–52 (2010).

[3] Bond, M., Coons, K. and McKinley, K.: PACER: Propotonal detection of data races, *Proc. 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp.255–268 (2010)

[4] Chord: A Versatile Program Analysis Plathome for Java, available from ⟨http://pag.gatech.edu/chord/⟩(accessed 2011-10-01).

[5] Christensen, A.S., Moller, A. and Schwartzbach, M.I.: Precise analysis of string expressions, *Proc. International Static Analysis Symposium*, Lecture Notes in Computer Science, Vol.2695, pp.1–18 (2003).

[6] Engler, D. and Ashcraft, K.: RacerX: Effective, static detection of race conditions and deadlocks, *Proc. 19th ACM Symposium on Operating Systems Principles*, pp.237–252 (2003).

[7] Grove, D., DeFouw, G., Dean, J. and Chambers, C.: Call graph construction in object-oriented languages, *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.108–124 (1997).

[8] Java String Analyzer, available from ⟨http://www.brics.dk/JSA/⟩(accessed 2012-08-10).

[9] Kim, K.: Precise Data Race Detection in a Relaxed Memory Model Using Heuristic-Based Model Checking, *Proc. 24th IEEE/ACM International Conference on Automated Software Engineering*, pp.495–499 (2009)

[10] Liang, S. and Bracha, G.: Dynamic Class Loading in the Java Virtual Machine, *Proc. 13th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, pp.36–44 (1998).

[11] Livshits, B., Whaley, J. and Lam, M.: Reflection analysis for Java, *Proc. 3rd Asian Conference on Programming Languages and Systems*, pp.139–160 (2005).

[12] Matthew, S.H.: *Flow Analysis of Computer Programs*, Elsevier Science Inc. (1977).

[13] Milanova, A., Rountev, A. and Ryder, B.: Parameterized object sensitivity for points-to analysis for Java, *ACM Trans. Software Engineering Methodology*, Vol.14, No.1, pp.1–41 (2005).

[14] Naik, M., Aiken, A. and Whaley, J.: Effective Static Race Detection for Java, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'06*), pp.20–29 (2006).

[15] Pratikakis, P., Foster, J. and Hicks, M.: LOCKSMITH: Context-sensitive correlation analysis for race detection, *Proc. ACM SIGPLAN Conference on Programming*

Language Design and Implementation (*PLDI'06*), pp.320–331 (2006).
[16] Praun, C. and Gross, T.: Static conflict analysis for multi-threaded object-oriented programs, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'03*), pp.115–128 (2003).
[17] Praun, C. and Gross, T.: Object race detection, *Proc. ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pp.70–82 (2003).
[18] Ronsse, M. and Bosschere, K.: RecPlay: A fully integrated practical record/replay system, *ACM Trans. Comput. Syst.*, Vol.17, No.2, pp.133–152 (1999).
[19] Sawin, J. and Rountev, A.: Improved Static Resolution of Dynamic Class Loading in Java, *Proc. 7th IEEE International Working Conference on Source Code Analysis and Manipulation*, pp.143–154 (2007).
[20] Schonberg, E.: On-the-fly detection of access anomalies, *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation* (*PLDI'89*), pp.285–297 (1989).
[21] Vallee-Rai, R., Co, P., Gargnon, E., Hendren, L., Lam, P. and Sundaresan, Y.: Soot – A Java optimization framework, *Proc. 1999 Conference of the Centre for Advanced Studies on Collaboratives Research*, pp.125–135 (1999).

魏 偉

1983 年生．2007 年中華人民共和国河北大学卒業．ソフトウェア開発会社勤務を経て，2012 年埼玉大学大学院理工学研究科修士課程修了．ソフトウェア検証の研究に従事．

吉浦 紀晃 （正会員）

1968 年生．1991 年東京工業大学工学部情報工学科卒業．1997 年同大学大学院博士課程単位取得退学．博士（学術）．東京工業大学助手，群馬大学助教授を経て，現在，埼玉大学大学院理工学研究科准教授．ソフトウェア検証やネットワーク運用技術の研究に従事．社会情報学会，IEEE-CS，ACM 各会員．