情報処理学会論文誌　Vol. 0　1–??? (??? 1959)

Traceglasses: 欠陥の効率よい発見手法を実現するトレースに基づくデバッガ

櫻井孝平[†1]　増原英彦[†2]　古宮誠一[†1]

現状のデバッグはプログラムの不正な動作を理解するために多くの時間とコストを必要とする．本研究は実行トレースを使った新たな欠陥の発見手法を提案することでデバッグの効率化を図り，トレースに基づく Java のためのデバッガである Traceglasses として実装する．Traceglasses は，プログラムの実行中のイベントを記録したトレースの対話的な表示と検索が可能で，利用者がプログラムの動作を遡って欠陥を発見することを支援する．Traceglasses は，トレースをメソッド呼出し関係により木構造に変換し，ブレークポイントデバッガに見られるようなステップ操作を行わずに，トレースの表示を行う．また，指定されたオブジェクトに対する操作のみを抽出したトレースを表示できる．これらの特徴により，利用者は，実行順に並んだ膨大なトレースに対して局所的および大域的に，必要な部分だけを理解し，効率よく欠陥を発見できる．Traceglasses の実装は，木構造を持つ膨大なトレースを効率よく探索し表示するために，専用のデータ構造を構築する．本発表では，Traceglasses の実装により実際のテキスト整形プログラムの欠陥をプログラムの出力から遡り発見する事例を説明し，提案した欠陥の発見手法と実装について実用上の有用性を示す．

Traceglasses: A Trace-based Debugger for Realizing Efficient Navigation

Kouhei Sakurai ,[†1] Hidehiko Masuhara [†2]
and Seiichi Komiya[†1]

Debugging requires considerable time and effort to understand incorrect behavior of a program. We developed a trace-based debugger called Traceglasses for Java. It provides the programmer a better means of navigating an execution history so that he or she can efficiently spot a defect in the program. First, the developers can browse an execution trace namely a sequence of events in the execution, with a view of local trace information, and a function to jump to a distant point in the trace. A local trace information is shown as a tree structure of method call, which liberate the programmers from stepped program executions, which are inevitable with breakpoint-based debuggers. In order to globally jump to a distant point in a trace the programmer can restrict a trace view to an object. Our implementation achieved efficient recording and displaying of traces by designing a dedicated data structure. In this presentation, we demonstrate usefulness of our debugger through a case study that is to find a defect in a practical text-formatiing program.

1. はじめに

ソフトウェアの正しさを向上する技術には，コードレビュー，モデル検査，テストおよびデバッグなどが挙げられる．現実のソフトウェア開発ではテストとデバッグが頻繁に行われる．開発者はテストによってプログラムの欠陥の存在を知ることができるが，欠陥の箇所を正確に知ることはできない．

デバッグはプログラムの欠陥 (defect) すなわちコード上の誤りを，プログラムの実行結果から発見する作業である[21]．欠陥を含んだプログラムの実行は故障 (failure) として開発者に観測される．そのような故障となる実行は，欠陥による振舞いによって状態が不正になり，その不正な状態が伝搬するように振舞うことで，不正な実行結果に至る．開発者は，欠陥を発見するために，そのような不正な実行結果から欠陥の振舞いに到達する必要がある．

本研究ではタイミングなどによらない，再現性のある故障を対象とする．そのような故障を引き起す欠陥は，実行時のデータと制御の流れを, 不正な実行結果に至る振舞いとして追跡することで発見できる．開発者は，まずプログラムの故障となる実行の，不正な実行結果に直接関わる時点（大抵は結果を出力する時点）の動作から追跡を開始する．この段階では，実行結果の出力に利用されるデータとそのデータに関連する制御の流れを追跡する．この追跡は，プログラムの実行全体のうち，特定の実行時点に対して局所的になる．次に，出力に利用されるそれらのデータがどの時点で生成されたかを，プログラムの実行全体にわたって大域的に追跡する．不正なデータが生成された時点を発見し，その生成に利用される他のデータや制御の流れを追跡する．この追跡も，生成の時点に対して局所的になる．各段階において，開発者は複数のデータや制御の流れの中から，不正とみなして追跡するものを

†1 芝浦工業大学 大学院工学研究科
Graduate School of Engineering, Shibaura-Institute of Technology
†2 東京大学 大学院総合文化研究科
Graduate School of Arts and Sciences, University of Tokyo

理解し選択する．これらの手順を繰り返すことで，いずれ最初に不正な状態となる欠陥の振舞いに到達し，欠陥を発見することができる．

デバッグを支援するツールであるデバッガは，様々な表示によってプログラムの実行に関する情報を利用者に表示することで，デバッグの効率の向上を図る．開発者はデバッガを利用し，不正な実行結果に関するデータや制御の流れの情報を対話的に得ることができる．デバッガは, ブレークポイントデバッガや逆戻りが可能なトレースに基づくデバッガなどの種類に分類できる．

デバッガの中でも広く普及しているブレークポイントデバッガは，プログラムの実行に沿った局所的な追跡であるステップ操作を基本とする．このため，これらのデバッガではデバッグに必要な情報を効率良く得られない．これらのデバッガはプログラムの実行を特定の時点で停止し，その時点の状態やメソッド呼出しの文脈を表示する．利用者の操作は，停止した時点での局所的かつ実行順に限定された追跡になるため，プログラムの実行を遡った大域的な追跡を必要とする欠陥の発見には不向きである．

トレースに基づくデバッガ[⋆1]は，プログラムの実行のすべてのイベントをトレースとして記録することで，大域的なデータと制御の流れの追跡を可能にする．ここでトレースとは，網羅的な実行時のイベント列とそれらの時点に関連するデータ[⋆2]をいう．これらのデバッガは，プログラムの要素 (変数やメソッド) に対して関連するトレースの表示が可能で，利用者は特定の不正なデータや制御の流れを容易に追跡することができる．一方，現状のトレースに基づくデバッガによるデバッグの手法は充分に確立されているとはいえない．そのため，多くのデバッガは局所的な追跡のために，ブレークポイントデバッガに似せたステップ操作とそのための表示を中心にしている．そのような表示を行うためのユーザーインターフェースは，利用者に不正な実行結果の追跡のために多くの操作を要求する．これは単に表示上の問題だけではなく，トレースの記録の形式など，デバッガの基本的な設計に関わる本質的な問題である．

本研究はトレースに基づくデバッガのためのトレースの内部表現形式と局所的および大域的な追跡を効率的に行う手法を新たに提案する．トレースの表現形式は，メソッド呼出しの関係に基づく木構造からなる．各ノードは値を伴った式として表示する．利用者は，このトレースの木のうえでの局所的な追跡によって，ステップ操作から解放される．さらにデバッガは，様々な手段による検索から得られる部分トレースを表示し，プログラム全体のトレースの木に対応させることで大域的な追跡を容易にする．また，トレースを変換して表示する機能により，利用者はトレースを容易に理解できるようになる．

本研究では，提案したデバッグ手法を可能にするインターフェースを実装した，Java のための高性能なトレースに基づくデバッガ Traceglasses を開発する．Traceglasses はトレースを木構造として効率よく表示する．

Traceglasses の実装を用いて，本研究の提案の有用性の評価実験を行った．Traceglasses を実際のオープンソースプログラムのテストケースに適用することで，速度と追跡手法に関する実現可能性を確認した．この実験からは，実際の欠陥の発見に利用できることが分った．

本論文の以降の構成は，まず 2 節では欠陥の発見の手順と，その際に必要な情報を例を使って示し，必要な情報の取得に対する既存のツールの問題点を説明する．3 節では提案するデバッグの手法を説明する．4 節は 3 節の手法に基づくデバッガ Traceglasses の利用と実装について述べる．5 節は評価実験について説明する．6 節は関連研究との比較について述べる．7 節は現在の Traceglasses の制限と課題について述べる．8 節はまとめである．

2. 追跡によるデバッグとその問題点

本節では，再現性のある故障を引き起す欠陥のデバッグにおける，追跡の手順と問題点を説明する．デバッグを効率よく行うためには，データと制御の流れの追跡に必要な情報が得られなければならない．開発者は，不正なデータと関連する制御の流れを，複数の候補の中から選んで追跡を行う．また多くの場合，追跡は実行とは逆順に行う必要がある．そのため，実際に流れるデータの値など，判断の材料となる実行時の情報が，実行順序や実行時点に制限されずに得られなければならない．

以降は単純化した例を使って，追跡の手順と必要となる情報および既存の手法の問題点を示す．まず，2.1 節で例のプログラムとその故障について説明する．続く 2.2 節ではブレークポイントデバッガでの手順と問題点を説明し，その後，2.3 節ではトレースに基づくデバッガでの手順と問題点を説明する．

2.1 例: テキスト整形プログラム

ここではデバッグの問題を説明するための簡単な例として，テキストを行ごとに整形するプログラムの欠陥を考える．このプログラムは各行にある連続した空白を一つにまとめる．ただし，行頭の空白はインデントと見なしそのまま出力する．なお，より実践的な実例は 5.2 節で示す．

⋆1 全知の (omniscient) デバッガなどとも呼ばれる．
⋆2 記録されるデータの具体的な定義はデバッガにより異なる．

```
1 class Tidy {
2   String[][] data;
3   void read(String text) {...}
4   void format() {...}
5   String concat() {...}
6   public String translate(String t) {
7     read(t);
8     return concat();
9 } }
```

図 1　クラス Tidy の定義
Fig. 1　The definition of class Tidy

例えばこのプログラムは期待される実行として，以下の左の入力から右の出力を行う (␣は空白を表す．).

入力	出力
test	test
␣␣foo␣␣bar	␣␣foo␣bar

デバッグ対象のプログラムが図 1 のようなクラス Tidy だとする．Tidy は整形を行う translate メソッド (6 から 9 行目) とそこから呼ばれる他の複数のメソッド (3 から 5 行目)，および内部的な状態であるフィールド data (2 行目) からなる．図では translate 以外のメソッドの定義は省略している．

このクラス定義が，メソッド定義のどこかに含まれる欠陥によって，以下のような不正な出力を行ったとする． インデントが単一の空白になり，他の複数の空白がそのまま出力されている．

入力	不正な出力
␣␣foo␣␣bar	␣foo␣␣bar

以降，この欠陥について既存のデバッガを使った場合のデバッグについて説明する．

2.2　ブレークポイントデバッガによるデバッグ

一般的なブレークポイントデバッガは，プログラムの実行をブレークポイントと呼ぶ特定の時点で停止させ，ステップ操作を行うことで，追跡に必要な情報を得ることができる．Tidy の例では，利用者は translate メソッドの開始時点 (図 1，7 行目) にブレークポイントを設定し，最初の read の呼出しについてステップ操作を行う．concat の直前 (8 行目) まで進めることで，計算が正しく行われたかを状態 data の内容をみて判断する．

ブレークポイントデバッガは基本的に停止した時点での局所的な情報しか得られず，プログラムの実行を逆に戻ることができないため，デバッグの効率が低下する場合がある．例えば，concat メソッド定義の後方に欠陥が含まれていた場合，最初からステップ操作を行う追跡では，利用者が欠陥に到達するまで多くの操作を必要とする．また，適当なブレークポイントの設定は，利用者の事前のプログラムの理解や勘に頼ることになり，困難である．

2.3　トレースに基づくデバッガによる逆戻りデバッグ

トレースに基づくデバッガは，デバッグ対象のプログラムの実行履歴を記録することで故障を再現する．これらのデバッガが記録する実行履歴はプログラム実行の網羅的なイベント列と関連するデータであり，トレースと呼ぶ．例えば，イベントにはメソッド呼出しやフィールドの代入が含まれる．プログラムのコード挿入などによって得られるトレースを，なんらかの形式でストレージに記録する[⋆1]．デバッガは記録したトレースからメソッド呼出しの文脈やオブジェクトの状態を復元し，表示を行う．

トレースに基づくデバッガは，ブレークポイントデバッガと異なり，任意の時点に関して実行順序に関わらずトレースに関する情報を表示できる．そのような表示を利用したデータと制御の流れの追跡は，多くの場合実行とは逆順となるため，**逆戻りデバッグ**と呼ぶ．以降，2.3.1 節では，例に対してトレースに基づくデバッガを利用した場合の，効率の良い逆戻りデバッグを追跡の段階ごとに説明する．この説明によって，効率の良い追跡に必要な情報と，その表示方法を明らかにする．その後，2.3.2 節では，既存のトレースに基づくデバッガの表示に関する問題点を述べる．

2.3.1　逆戻りデバッグでの追跡

効率の良いデバッグのための，データと制御の流れの追跡は，プログラムの実行順に関係なく，イベント列に対して局所的または大域的に，段階的に繰り返す必要がある．

- **局所的な追跡**は，プログラムの特定の時点の前後のデータや制御の流れに対して行う．

⋆1 実装によってはメモリ上に直接記録するものもある．

これは，連続した複数のメソッド呼び出し間で受け渡されるデータを追跡することも含む．局所的な追跡によって，開発者は不正なデータとそれに関わる計算を理解し，欠陥と直接対応する振舞いを発見するか，または更なる追跡のための他の流れを選択する．典型的には，ある時点の計算が，直前で生成された他の複数のデータを利用する場合に，それらのデータのうち不正なものを選択し，局所的な追跡を行う．

- **大域的な追跡**は局所的な追跡で発見した不正なデータや，それらに関わるプログラムの要素 (メソッドやフィールド) から，関連する別の時点へ大域的に移動するために行う．例えば，あるデータに関わる要素には，そのデータを値として保持するフィールドが考えられる．そのフィールドを読書きする時点は，不正なデータを生成する時点と利用する時点を含み，それらが全体のイベント列から見て離れている場合，大域的に追跡する必要がある．大域的な移動の後には局所的な追跡に移る．

これらの追跡は多くの場合，対象となる流れの候補を開発者が選択する必要があるため，対話的である．

トレースに基づくデバッガは，追跡のためにトレースと対応するソースコードの情報を表示する．トレースには，例えばメソッドに渡される値などの，その時点で利用されたデータの値が伴う．

例えば Tidy に対しては以下のような段階によってデバッグが進む．

段階 a: 開発者は，まず translate が返す文字列が不正であることを理解し，局所的にその文字列の生成過程を追跡する．その結果，文字列が concat で実行される以下のようなトレースに由来することが分る．

```
translate("␣␣foo␣␣bar")
 return concat()
  return buf.toString();
  //不正な結果"␣foo␣␣bar") を返す.
```

これらのトレースはメソッドの呼出し関係に基づき，translate の結果が concat の呼出し結果であり，concat の結果が buf に対する toString の呼出しの結果であることを意味する．

段階 b: concat メソッドでは文字列バッファである StringBuilder のオブジェクト buf によって文字列を生成する． 大域的な値の追跡によって，この文字列バッファに関連する以下のようなトレース列が得られる．

```
StringBuilder buf = new StringBuilder();
buf.append("␣"); //(1)
buf.append("foo");
buf.append("␣␣");
buf.append("bar");
buf.toString(); //" foo bar"を返す.
```

これらのトレースの列において，buf はすべて同一のオブジェクトであるとする．このトレース列から，(1) に注目すると，この時点ですでに不正な文字列が追加されることがわかる．

段階 c: 前の段階の (1) の前後のトレースにおける局所的なデータの流れに注目し，以下のような連続したトレースを得る．

```
s = data[0][0]; //s=="␣"となる.
buf.append(s); //(1)
```

上記のトレースから配列 data の要素 [0][0] に格納される値が不正であることがわかる．

段階 d: data[0][0] に対する書き込みを大域的に検索し[*1]，結果から書き込みの時点に移動する．

段階 e: 移動した時点から制御の流れを局所的に追跡する．その結果，concat の前方で呼ばれる format メソッドに，以下のようなコードによるトレースがあることが分る． このコード中のインデント部分を省くための条件 j==0 が，正しくは j!=0 であるような欠陥であると判明する．

```
if (j==0 && spaces(data[i][j])) {
 data[i][j] = "␣";
}
```

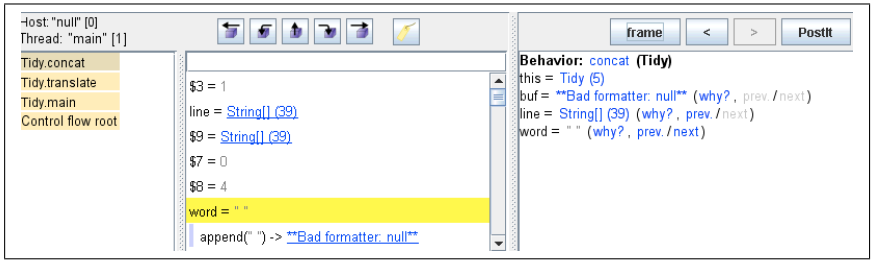


図 2　TOD[14] による追跡の例: 各表示は，左はメソッド呼出しスタック，中央は選択されたメソッド内部のトレースのリスト，右は選択されたトレースの時点の状態を表す．

Fig. 2　An example of navigation by TOD[14]: The left view is a call stack. The center view is a list of internal traces of a selected method. The right view is states of a selected trace.

2.3.2　既存のトレースに基づくデバッガの問題点

既存のトレースに基づくデバッガの実装はなんらかの手段で大域的および局所的な追跡を支援するが，必ずしも効率が良いとはいえない．具体的には，ODB[11] や TOD[14] は，前節で示した**段階 a** のような，メソッドをまたぐデータの受け渡しなどが把握しにくい．これらのデバッガは選択したトレースに関する値の表示とステップ操作を提供する．しかし，そのような表示からは，利用者は局所的なデータと制御の流れの変化の情報を一度に得ることができず，理解に時間を要する．また，大域的な追跡のための検索の多くは，結果を一覧できず，検索結果に対して順に局所的な追跡に移行する必要がある．

例えば TOD では図 2 に示すように，左側のメソッドの呼出しスタックの表示には値は現れず，また単一のメソッド中のトレースのみを中央に表示する．右端は選択したトレースの時点の状態のリストである．この表示からは，例えば `concat` の呼出しの結果の値が，どのように呼出し元の `translate` に流れるかがわからない．また，トレース表示に配列の操作や数値の演算が含まれない．このため，例えば文字列バッファに追加される空白がどのように得られたかが不明である．

さらに既存のトレースに基づくデバッガは，トレースの表示形式が固定されるため，利用者の理解を妨げる場合がある．例えば，文字列を内部表現として文字コードで扱うような場合は，トレースの表示を変更し，利用者に理解しやすい文字での表示を提供すべきである．

⋆1 実際には配列は各次元ごとにオブジェクトとして扱われる．配列に適当な ID 番号が降られることで，複数のトレース間で同定する．

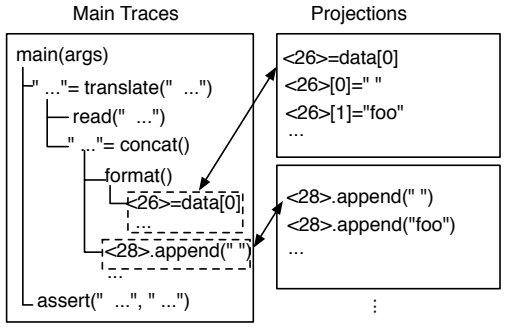


図 3　トレースを利用したデバッグにおける追跡のための表示

Fig. 3　Views for navigation of trace based debugging

このようなトレースの表示は，トレースに基づくデバッガにとって基本的な設計の決定に基づく．基本的な表示に関わる設計は，デバッガにとって本質的なものであり，デバッグの効率に影響する．例えば，TOD のようなスケーラブルな実装では，ユーザーインターフェースの応答性能を向上させるために，表示を左側のメソッドの呼出しスタックと，そこで選択したメソッドに関するトレースのイベント列に限定していると推測できる．また，数値の演算など特定の実行情報をトレースとして記録しないことも，同様に効率のためといえる．

既存の実装が，局所的および大域的な追跡を妨げる設計を選択したのは，2.3.1 節で示したようなデバッグの手法が確立していないためといえる．

3.　提案する欠陥の発見手法

本研究は，トレースに基づくデバッガのための，新たな欠陥の発見手法を提案し，Java 上の実装 Traceglasses を開発する．提案する欠陥の発見手法は，2.3.1 節で示した逆戻りデバッグでの追跡手法に対応し，2.3.2 節で示した問題点を解決する．この提案は，追跡のためのトレースの表示手法と，その追跡手法，および表示のためのトレースの表現形式からなる．まず 3.1 節で，欠陥の発見手法を説明する．続く 3.2 節で，提案する手法で扱うトレースの表現形式を定義する．3.3 節ではこの表現形式を使った検索と変換について説明する．

3.1　提案する欠陥の発見手法

本研究で開発する Traceglasses による追跡のための表示は，図 3 のように抽象化できる．図の実線の矩形はトレースの表示を表す．

- 左側は**プログラム全体のトレース**の表示である．プログラム全体のトレースは，スレッドおよびメソッド呼出し関係により，木構造をなす．図の各行はトレースの各イベントであり，それらを接続する実線と字下げによって木構造を示している．トレースの各イベントは値を伴った式として表示され，メソッド間での値の受け渡しによるデータの流れを理解することができる．値の表示形式は型によって異なるが，いずれも各イベントの式の行に収まるように表示される．
- 右側の表示はプログラム全体のトレースを投影した**部分的なトレース**の表示であり，対話的な検索の結果などを表示する．表示内容は検索結果により生成されるため，トレースの表示は動的に増加する．図中の矢印で示すような左右のトレースのイベントの対応付けから，右側で選択したイベントと関連する左側のイベントを選択することができる．
- 左右の表示で選択したトレースのイベントに対して，対応するソースコードの位置情報を得ることができる． 利用者はソースコードから追跡に必要な情報を得ることができる．

本研究の提案する新しい欠陥の発見手法は，同一のトレースの形式を，局所的および大域的な追跡のための表示で統一的に利用することで，利用者が効率良くデバッグに必要な情報を得ることを可能にする．トレースは，すべての計算に関して簡潔で表現力のある形式として定義を与え (3.2 節で述べる．)，表示のみならず検索においても同じ表現をそのまま利用できる (3.3 節で述べる．).

上記の複数のトレースの表示により，局所的および大域的な状態と制御の流れの追跡を実現する．

- **局所的な追跡**のためには，プログラム全体のトレースの表示から，注目するイベントの前後を閲覧し，データと制御の流れを理解する．制御の流れは，メソッド呼出しの関係と，細かい粒度の (命令またはコード上の文に対応した) メソッド内部の制御の流れからなる．データの流れは，メソッド間の値の受け渡しなどの，メソッド内部で利用された値の表現と，それらにラベル付けされたローカル変数名からなる．また，選択したトレースのイベントに対応するソースコードの表示によって，局所的な追跡をより容易に行うことが可能になる場合がある．局所的な追跡のための操作として，例えば 2.3.1 節の**段階 a**，**段階 c** および**段階 e** での操作を実現する．
- **大域的な追跡**は，投影されたトレースの表示の生成によって行う．典型的には，投影のトレースはプログラム全体のトレース上から局所的な追跡で判明したプログラムの要素 (値やメソッド名など) から，トレースを検索することで生成する．例えば，ある時点のイベントのオブジェクトについて逆戻り (プログラムの実行とは逆順) に検索を行うことで，そのオブジェクトの操作の履歴をトレースのイベント列として得ることができる．具体的には 2.3.1 節の**段階 b** および**段階 d** の操作を実現する．

これらの追跡は，図 3 で示す表示において，利用者は左右交互にトレースの表示を移動することで行う．実際のツールでの表示は 4 節で示す．

$$
\begin{aligned}
s &::= r = e \;\mid\; c \;\mid\; n \\
&\quad \mid \texttt{return}\ v \;\mid\; \texttt{throw}\ v \\
&\quad \mid \texttt{if}(a) \;\mid\; \texttt{goto} \;\mid\; \texttt{thread-}i \\
e &::= c \;\mid\; n \;\mid\; a \;\mid\; r \;\mid\; v \\
c &::= T.m(v\ldots) \;\mid\; v.m(v\ldots) \\
n &::= \texttt{new}\ T(v\ldots) \;\mid\; \texttt{new}\ T[v]\ldots \\
a &::= uop(v) \;\mid\; v\ bop\ v \;\mid\; v\ \texttt{instanceof}\ T \\
r &::= v \;\mid\; T.f \;\mid\; v.f \;\mid\; v[v] \\
v &::= label{:}l \;\mid\; l \\
l &::= \texttt{<}i\texttt{>} \;\mid\; \texttt{"}str\texttt{"} \;\mid\; p
\end{aligned}
$$

図 4　トレースの構文
Fig. 4　Syntax of traces

3.2　トレースの表現形式

提案手法では，トレースの表示のために木構造を持つトレースの表現形式を利用する．木のノードとなるトレース中の単一のイベントは，具体的な値を伴なうプログラムの要素の式として表現する．例えばメソッド `toString` の呼出しとその結果を変数に代入するようなイベントは以下の式で表現できる．

```
s:"foo" = <21>.toString()
```

ここで `s` はローカル変数名であり，代入の結果，値が`"foo"`となったことを意味する． また代入文の右辺の`<21>`はオブジェクトの ID 番号による表現である．

具体的なトレースの表現形式は，図 4 のような構文として定義する．

- s はノードであり，基本的にスタック操作などを除いた Java のバイトコード 1 命令に対応する．$r = e$ は左右に置かれる要素によって右の値を左の対象に代入する代入文か，右の式の結果が左である文のどちらかを意味する．`if` は条件分岐の通過を意味する． 同様に `goto` はジャンプ命令の通過を意味する． `thread-`i はスレッド i のルート

ノードを意味する．

- e は式を表す．
- c はメソッド呼出しである． m はメソッド名である． 静的なメソッドの場合は型名 T を伴ない，インスタンスメソッドの場合は対象の値 v を伴なう． 値を返す場合，代入文の右側に配置され，左側は生成されたオブジェクトとなる．
- n はオブジェクトまたは配列の生成を表す． オブジェクトの生成は new 命令とコンストラクタの呼出し命令からなる． 代入文の右側に配置され，左側は生成されたオブジェクトとなる．
- a は基本的な演算を表す． *uop* は単項演算子，*bop* は二項演算子である． 代入文の右側に配置され，左側は演算の結果となる．
- r はフィールド，ローカル変数または配列の利用を表す． 代入文の左側に配置された場合，右側の値の代入の操作を意味する． 右側に配置された場合は，参照の結果，左側の値になることを意味する．
- v は値の表現であり，ローカル変数を介す場合は *label* を伴なう．
- l はオブジェクト i の参照，文字列オブジェクトの参照 *str*，またはプリミティブ値 p の表現を意味する．

この形式では，トレースの各イベントはスタックマシンである Java バイトコードの命令とは直接対応せず，Jimple[18] のような三番地形式に基づく．

トレースの各イベントはこの式表現に加えて，内部的に対応するソースコードのファイル名と行番号を持つことができる．

3.3　トレース式による検索と変換

Traceglasses は定義したトレースの表現形式によってパターンマッチを行う検索が可能である．例えば，`"foo"`を返す任意のオブジェクトに対する `toString` の呼出しは以下のようなパターンの式として記述できる．

```
"foo" = ?v.toString();
```

?v は自由変数であり，任意の式を表す．この機能により，2.3.1 節の段階 d の配列に対する書き込みの検索を以下のような記述で実現できる．

```
<19>[0] = ?v;
```

ここでは配列オブジェクトが`<19>`であるとする． 配列オブジェクトの値は追跡によって判明するので具体的な値をパターンに含めることができる．Traceglasses はマッチした結果のトレースの列を，部分的な投影のトレースのイベント列として表示する．

パターンのマッチはトレースの各イベントに対して行われるが，イベント中の式に対してマッチを行いたい場合は `$e(`e`)` という式を利用することができる．例えば，`<19>`の配列の 0 番目を利用するイベントの検索は以下のように書ける．

```
$e(<19>[0]);
```

本研究の提案する追跡手法はトレースの式を定義された表現形式による異なる別の式に変換することを可能にし，利用者の理解を容易にする．トレースの表示ごとに適用される，なんらかの変換ルールによって，その内部のトレースの式を別のトレースの式に変換する．

例えば，プログラムが文字を `int` 型の文字コードで扱う場合は，文字コードを表示可能な文字に置き換えることで，利用者はトレースの理解が容易になる．変換は以下のような変数束縛とスクリプトを埋め込んだ規則の式として定義できる．

```
?o.append(?i) => ?o.append({:
  chr(i.getValue().toString())
:});
```

変換規則は s`=>`s`;` の形式によって記述し，左側のパターンにマッチしたトレースの式を右側のパターンに変換する．右側では埋込みスクリプト (`{:`から`:}`) が利用できる． スクリプトは動的な言語で記述され，パターンで束縛した変数を参照し，結果をトレースの式の値として表示する．現在の Traceglasses は，Java で記述された Python の処理系である Jython[*1] に対応しており，例えば上記の例では `i` に束縛された式を数値の文字列として参照し，Python の組込み関数 `chr` によって文字の表現に変換する．この変換規則は例えば以下の左側を右側に変換する．

変換前	変換後
`<1>.append(65)`	`<1>.append("A")`

4.　Traceglasses による実装

前節の欠陥の発見手法を実現するために，本研究ではトレースに基づくデバッガ Traceglasses を Java で実装した[*1]．コードサイズは約 2 万行からなる．まず，4.1 節で概要としてデバッガの利用方法を説明する．続く 4.2 節ではトレースの記録方式を説明する．その後，4.3 節ではトレースの表示の実装方法について説明する．

---

*1 `http://www.jython.org/`

*1 実装は `http://www.komiya.ise.shibaura-it.ac.jp/project/td/traceglasses/` で公開する．

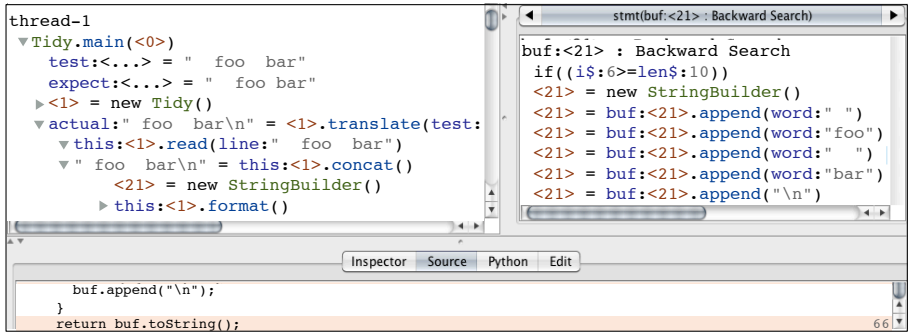


図 5　Traceglasses のトレース表示部分
Fig. 5　View parts of traces in Traceglasses

4.1　概　　要

Traceglasses はデバッグ対象の Java プログラムの実行をトレースとして記録し，利用者が対話的な欠陥の追跡に利用するため，画面上のユーザーインターフェースにより表示を行う．トレースの記録は，デバッグ対象のプログラムに記録のためのコードを埋込むことで実現する．

図 5 は Traceglasses の画面の表示部分である．上部左側の領域はプログラム全体のトレースの表示である．右側の領域は投影による部分的なトレースの表示である．下部は対応するソースコードの表示である．

図 5 の左側のプログラム全体のトレースの表示は，2.3.1 節の**段階 a** で利用する場合の例である．Swing の木構造表示のための GUI 部品である JTree を利用して，折畳みによって木の部分の表示が省略される．

図 5 の右側の部分的なトレースの表示は，2.3.1 節の**段階 b** でバッファ<21>に対して検索を行った場合の例である．検索は，選択したトレースのイベントに対するポップアップメニューから，イベント中の値と検索手法を選ぶことで実行できる．例えば図 6 の上部は，2.3.1 節の**段階 c** で，バッファに追加する文字列を配列から参照するイベントを選択している．このイベントに対してポップアップメニュー (図の下部) から配列<19>に関する検索を選択することができる．また，3.3 節で説明したトレース式の記述による検索と変換は，画面下部のタブ表示を切替えることで可能になる．検索は値に関して逆戻りまたは実行順に実行できる．

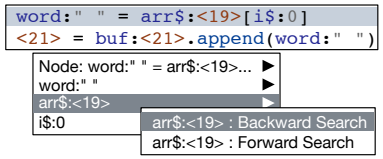


図 6　局所的な追跡のためのトレース表示 (上) と検索のためのメニュー (下)
Fig. 6　Trace view local navigation (the upper part) and menus for searching (the lower part)

4.2　トレースの記録

Traceglasses はトレースの記録のために， BCEL[⋆1]を利用してコードの挿入を行い Java のクラスファイルを改変する．図 7 はトレースの記録の行程を示している．まず図の (1) で示すコードの挿入時に，対象のクラスファイルにトレースの記録のためのコードの挿入を行う (図の Instrumented classes)．同時に静的な情報を保持したファイルを生成する (図の Trace expression information)．図の (2) に示すトレースの記録は，コードの挿入が行われたクラスを実行することで，トレースの木構造を保存するファイル (図の Trace tree information) と，各トレースの情報 (伴う値と対応するトレース式のインデックスからなる，図の Traces information) の出力を行う．

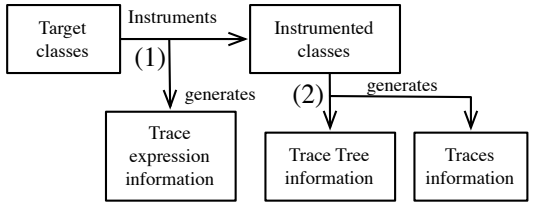


図 7　トレースの記録の行程
Fig. 7　Processes of recording traces

3.2 節で述べたように，トレースの表現形式は Java バイトコードの複数の命令を組合わせた三番地形式から生成する．BCEL はそのような形式を持たないため，三番地形式を持つバイトコード操作フレームワークの Soot[18] と同様に，Traceglasses のコード挿入時に読込んだクラスファイルのバイトコードの手続き内部のデータの流れを解析し，三番地形式へ

⋆1 http://jakarta.apache.org/bcel/

変換する[⋆1]．

静的な情報には，コード上の位置を示す固有の番号に対応したトレース式と，ソースファイル名および行番号が含まれる．

トレースの木構造は表 1 のような要素を持つレコードの列によって保存される．レコードの各要素はすべて同じ幅の整数値であり，現在の実装では 32 ビットの Java の int として扱うため，各レコードは 24 バイトのサイズになる．trace_index は各トレースの情報を格納するファイルの番地を指している． その他の要素のノード番号は木構造のレコードの番号を意味する．trace_index が指す各トレースの情報は，伴う値の数によって可変長である．

表 1　トレースの木構造のためのレコードの要素
Table 1　Fields of the record for tree structure of traces

要素の名前	説明
trace_index	トレースの情報の番地
prev_sibling_node	同一階層の前のノード番号
next_sibling_node	同一階層の次のノード番号
parent_node	親の階層のノード番号
first_child_node	子の階層の最初のノード番号
last_child_node	子の階層の最後のノード番号

トレースの木構造およびトレースの各イベントの情報は，プログラムの実行時に漸進的に生成される．図 8 は木構造を表 1 のレコードとして記録するためのアルゴリズムである．write($trace_t, trace_m, trace_i, type_i$) を対象プログラムの特定の各イベント実行時に呼びだすことで記録が行われる．ここで，$trace_t, trace_m, trace_i$ はそれぞれ実行中のスレッド，実行中のメソッド，イベントのトレース番地である．$type_i$ はイベントの種類を示し，BEFORE, AFTER または POINT のどれかをとる． BEFORE，AFTER はメソッド本体の実行の前後であり，POINT はそれ以外の命令によるイベントである． イベントがメソッド呼出しである場合は，1 つの $trace_i$ に対して，BEFORE と AFTER に対応する 2 回の write の呼び出しが起こる．

記録のために，メソッド呼出しスタックと対応するノード番号のスタックを利用する．write の 2 行目の $stack[trace_t]$ は対応するスレッドのスタックを取得する操作である．

3 から 6 行目はスタックが空であった場合に，新たにスレッドのノードを作る操作を行う．add_child(n, t) はノード n に対して，trace_index が t となる新たな子ノードを追加して，そのノード番号返す関数である．$root$ は木構造全体のルートとして用意されるノードである．push(s, n) はスタック s に n を追加する操作である．

7 から 10 行目はメソッドのノードが存在しない場合に，そのノードを作る操作を行う．top(s) はスタック s の先頭の要素を返す関数である．$n.f$ は n 番目のノードのレコードの要素 f に対する参照である．

11 から 15 行目はイベントのノードを追加する操作を行う．一方，16 行目から 20 行目はメソッドの終了時にスタックから要素を削除する操作を行う． pop(s) はスタック s の先頭を削除する操作である．削除は例外による大域脱出後のメソッドの終了を考慮し，スタックの先頭の要素が対応する BEFORE のイベントによって追加されたノードになるまで繰返す．

22 行目以降の add_child はノードを追加する関数のアルゴリズムである．23 行目の create_record() はメモリマップ上に新たに図 1 で示したレコードを追加し，レコードのノード番号を返す操作である．レコードの要素の値は，未定義値として-1 で初期化される．25 から 32 行目の親子間でのレコードの要素の設定によって，ノードの追加を実現する．

トレースの情報には実行時の値が含まれる．実行時の値は Java の基本型，文字列 (String オブジェクト)，オブジェクト参照の ID 番号のどれかとして記録される．オブジェクト参照の ID 番号を得るためには，ID 番号を管理する表を実行時に構築する．この表の実装には以下の選択肢が考えられる．

(1)　実行時のオブジェクトをキーとした `IdentityHashMap` を使う．`IdentityHashMap` はキーをオブジェクトの参照等価性によって区別する．プログラム終了までにオブジェクトが解放されないため，メモリを圧迫するなどの問題が発生する場合がある．

(2)　`System.identityHashCode(`o`)` で得られるハッシュ値をキーとした `HashMap` を使う．ハッシュ値はメモリアドレスに基づく 32 ビットの値で，ID 番号の重複が発生する場合がある．

(3)　実行時のオブジェクトをキーとした `WeakHashMap` を使う．`WeakHashMap` はキーが弱参照として保持されるため，メモリを圧迫することがない．キーを `equals` メソッドにより区別するため，`equals` メソッドのオーバーライドによってオブジェクトの区別ができなくなる場合がある．

現在の実装では ID 番号の正確さを重視し、`IdentityHashMap` を選択している．これらの実装の切り替えは容易である．

⋆1 Traceglasses が Soot でなく BCEL を利用するのは，安定した実装のためである．

```
function write(trace_t, trace_m, trace_i, type_i)
 1: begin
 2:   s ← stack[trace_t]
 3:   if s = {φ} then
 4:     n_t ← add_child(root, trace_t)
 5:     push(s, n_t)
 6:   end if
 7:   if top(s).trace_index = trace_m then
 8:     n_m ← add_child(top(s), trace_m)
 9:     push(s, n_m)
10:   end if
11:   if type_i = BEFORE or POINT then
12:     n_i ← add_child(top(s), trace_i)
13:     if type_i = BEFORE then
14:       push(s, n_i)
15:     end if
16:   else if type_i = AFTER then
17:     while top(s).trace_index ≠ trace_i do
18:       pop(s)
19:     end
20:   end if
21: end
```

```
function add_child(n_p, trace)
22: begin
23:   n ← create_record()
24:   n.trace_index ← trace
25:   n.parent_node ← n_p
26:   if n_p.last_child_node = −1 then
27:     n_p.first_child_node ← n
28:   else
29:     n_p.last_child_node.next_sibling_node ← n
30:     n.prev_child_node ← n_p.last_child_node
31:   end if
32:   n_p.last_child_node ← n
33:   return n
34: end
```

図 8　漸進的に木構造を記録するアルゴリズム
Fig. 8　The algorithm for incrementally recording tree structure

4.3　トレースの表示

追跡のためのトレースの表示は，前節で示した記録された情報を段階的に読込むことで行う．読込みの効率を向上させるため，Java の NIO パッケージによって提供されるメモリマップ IO を利用する．

Traceglasses ではトレースの木構造の遅延読込みによって大規模なトレースを扱うことができる．全体のトレースの表示において，利用者が折畳みを展開した時点で，展開したノードの子要素をファイルから読込んで表示する．子要素の読込みは，表 1 で示した木構造の (メモリにマップされた) ファイルの探索によって行う．

```
function expand_trace(n_p)
1: begin
2:   l ← {}
3:   n_c ← n_p.first_child_node
4:   while n_c ≠ −1 do
5:     l ← l + get_trace(n_c)
6:     n_c ← n_c.next_sibling_node
7:   end
8:   return l
9: end
```

図 9　トレースのノードの展開を行うアルゴリズム
Fig. 9　The algorithm for expanding a trace node

図 9 はノード展開時にトレースを読込むためのアルゴリズムである．このアルゴリズムは n_p 番目ノードの子要素のトレースのイベント列を返す．返される集合 l はイベントの順序付き集合であり，$l + t$ は l の末尾にイベント t を追加する操作である．get_trace(n) は，n 番目のノードに対応するトレースの静的な情報と値の情報を読込み，トレース式として返す関数である．また，トレースの検索に対しても，表示と同様の木構造の探索を行う．

トレースの変換は，トレースの各イベントの表示時に動的に行う．表示されるトレースに対してのみ，マッチと変換が発生し，変換結果の式は描画に利用されるだけで破棄される．このため，変換によってツリーの全探索は行わず，また追加の大きなメモリ領域を必要としない．

5. 評　　価

実装した Traceglasses を実行性能の可用性と，提案した欠陥の追跡手法の有用性を実験により評価する．実行性能については，実行速度と必要とする記憶領域のサイズに関して計測する (5.1 節)．手法の有用性については，実際の欠陥に対して Traceglasses を適用した場合の追跡手順の例により示す (5.2 節)．

実験の対象として，Traceglasses を Java で記述されたオープンソースの HTML 整形プ

ログラムである jtidy[⋆1] と，同じく Java のビルドシステム apache-ant[⋆2](以下 ant) の実際の欠陥を発見するために適用した．これらの欠陥はバグデータベースにそれぞれ jtidy は ID763191，ant は ID41151 のバグ報告として登録されたものである．これらの欠陥は，次のような理由から実験の対象として適当であると判断した．1) バグ報告と再現可能な自動化されたテストプログラムが存在する．2) プログラムが小規模でない．3) グラフィカルなユーザーインターフェースをもたず，出力結果が画面上の表示や利用者の操作に依存しない．4) 短い追跡で簡単に発見できる欠陥でない．5) jtidy の欠陥については，バグ報告に対して修正が行われていないことも考慮した．なお，選択したテストケースのサイズの程度について，5.1 節の最後で追加的な実験とともに説明する．

いずれの実験も，MacOS X 10.6, CPU: Intel Core 2 Duo 2.4GHz, 4GB RAM, JVM1.6.0_15 (-Xmx256m) の環境で実施した．

5.1　実行性能に関する計測

トレースに基づくデバッグの記録や表示にかかる時間は，デバッグ対象のプログラムによって大きく変化するため，一般化することは困難である．本論文では，実際のプログラムでの場合による性能計測に基づき評価する．

jtidy と ant の欠陥の事例に対して，以下の観点から性能を計測した．

(1)　コード挿入の性能．結果は表 2 に示す．
(2)　実行時のトレースの記録に関する性能．結果は表 3 に示す．
(3)　デバッグ時の表示に関する性能．結果は表 4 に示す．

いずれの計測も，時間については 3 回の実行のうち最もよい値を採用している．

表 2 の (1) コード挿入の性能は，プログラム全体に対してコードを挿入する時間と，4.2 節で示した挿入時に生成される静的な情報のサイズを計測した．コード挿入時間の増加の要因はクラス数の全体のサイズだけでなく，大きなサイズのクラスにもある．jtidy は ant に比べクラス数は少ないが，サイズの大きいクラスの割合が大きく，クラスファイルの平均サイズでは ant より大きな値となっている．このためこれら 2 つのプログラムの挿入時間にはクラス数の倍率ほどの違いは出ていない．

表 3 の (2) 実行時のトレースの記録に関する性能については，それぞれバグ報告に対応するテストケースを実行した時の記録されるトレースのイベント数，記録によって発生する速度低下，そして記録によって生成される動的情報のサイズについて計測した．トレースを記録しない場合の本来のテストケースの実行時間は，それぞれ jtidy が 0.5 sec. ，ant は 0.6 sec. であった (JVM の起動時間を含む)．これらの結果から，おおまかに 1000 件のトレースごとに，約 20–30 msec. の時間と，約 30–40KB のストレージサイズを必要とすることがわかる．

⋆1 http://jtidy.sourceforge.net/
⋆2 http://ant.apache.org/

表 2　コード挿入の性能
Table 2　Performance of instrumenting code

プログラム	クラス数	クラス平均サイズ	挿入時間	静的情報のサイズ
jtidy	145	3.58KB	10.42 sec.	3.6MB
ant	752	3.19KB	26.48 sec.	14.0MB

表 3　実行時のトレースの記録に関する性能
Table 3　Performance of recording traces at runtime

プログラム	イベント数	速度低下	動的情報のサイズ
jtidy	98225	2.60 sec.	3.78MB
ant	191839	3.07 sec.	8.14MB

表 4 の (3) デバッグ時の表示に関する性能については，3.3 節で説明したようなパターンマッチによりトレース木全体を検索する時間を計測した．パターンの式は`$e("hello")` で，いずれの場合もどのトレースにもマッチしないが，記録された全てのトレースの探索が発生する．4.3 節で説明したように，検索によってトレースの木構造全体が探索され，メモリ上への読込みが発生する．このため記憶領域に余裕があれば，2 回目以降の検索は読込み済みの木構造に対する探索になり，初回に比べ高速になる．

いずれの場合でも，現在の実装ではトレースの記録には時間がかかるが，デバッグ時の利用に対して実用上は問題ないレベルであった．5.2 節で説明する探索の適用事例に際しても，追跡に支障をきたすような性能低下はなかった．今回の事例のように，自動化されたテストケースの実行であれば，プログラムは全体のうち固定された部分的な実行に限定されるため，性能に関する問題が起きにくいと考えられる．

テストケースのサイズと性能低下の傾向

jtidy の事例のテストケースについて，全体のうちどの程度の規模であるかを参考までに調べた．jtidy には 223 個のテストケースが付属しており，それら全てについて Traceglasses を適用してトレースを記録して実行する場合と，本来の (トレースを記録しない) 場合の実行時間を比較した．結果は，3 回の実行のうち最も小さい値を採用し，図 10 に示す分布と

表 4　デバッグ時の表示に関する性能
Table 4　Performance of viewing traces at debugging

プログラム	検索時間 (1 回目)	検索時間 (2 回目)
jtidy	1.7 sec.	0.6 sec.
ant	2.0 sec.	1.3 sec.

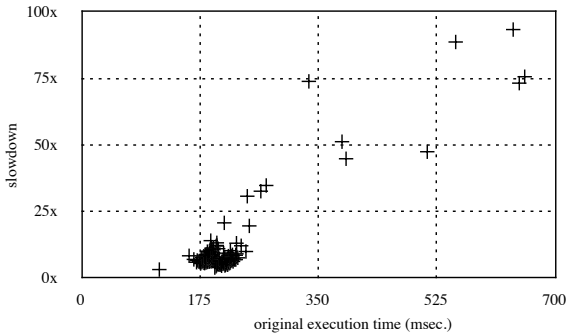


図 10　jtidy の 223 テストケースの本来の実行時間と
トレースを記録した場合の速度低下の分布
Fig. 10　Distribution of original execution time of jtidy's 223 test cases
and slowdown by recording traces

なった．図の横軸は本来のテストケースの実行時間を示し，縦軸はその時間に対してトレースを記録した場合の実行時間の倍率であり，トレース記録による速度低下を示す．なお 209 個のテストケースが，本来の実行時間 150–250 msec. かつ速度低下の倍率 4–14 倍の範囲に集中した．

最も速度低下が大きい場合は，本来 638 msec. の時間がかかるテストケースに対して，トレースを記録する場合は 59484 msec. (約 93 倍) であった．このような大きな低下は，大きなサイズのテストデータと多くのエラーログ出力のためである．一方，最も速度低下が小さい場合は，本来 115 msec. の時間がかかるテストケースに対して，トレースを記録する場合は 362 msec. (約 3 倍) であった．

事例として選択したテストケースは本来 191 msec. の時間がかかり (JVM の起動時間を含まない)，トレースによる速度低下の倍率は約 14 倍であった．テストケース本来の実行時間の全体の平均は 214.7 msec. ，また速度低下の倍率の平均は 9.62 倍であり，選択した事例は平均に近いものであった．

5.2　追跡手法の適用事例

欠陥を発見する実際の手順について，jtidy の事例 (5.2.1 節) と，ant の事例を含む他の事例 (5.2.2 節) について説明する．5.2.3 節ではまとめを述べる．

5.2.1　jtidy の事例

jtidy のバグ報告 (ID763191) は，HTML タグの属性に渡される文字列中の複数の空白を意図せず単一化するという故障の内容である．

欠陥の発見は，jtidy の内部のコードに対する知識を持たない第 1 著者が Traceglasses によって行った．発見手順を以下に示す．なお，以下に記載されるオブジェクトの ID 番号は実行する環境によって異なる可能性がある．

段階 1　テストケースの期待値と実際の値の比較を行う `assertEquals` の呼出しを，局所的な追跡により探す．期待値と結果の値は以下のような文字列である．

期待値	`"AB␣␣␣␣␣␣␣␣12345␣␣␣␣␣␣␣␣xy"`
結果の値	`"AB␣12345␣xy"`

段階 2　assertEquals に渡される実際の値を逆戻りに検索する．`<583>.toString()` が該当する．

段階 3　さらに`<583>`を逆戻り検索する．その結果

```
<590> = OutFactory.getOut(...,<583>)
```

という呼出しが該当し，結果を格納する`<583>`を包むオブジェクト`<590>`を作ることが分る．

段階 4　`<590>`を実行順に検索する．`<590>.outc(`c`)` という呼出しを多く含む 764 のトレースが該当する．c は文字コードであり，文字を順に出力することが推測されるが, 人間には出力の内容の理解が困難である．そのため，3.3 節で説明したように，以下のような表現形式の変換を行う．

```
<590>.outc(?c) =>
  <590>.outc({: chr(c.getValue()) :});
```

この結果，`outc("A")`，`outc("B")`，`outc(" ")` の順で出力を行うことが分る．

段階 5　`outc(" ")` のトレースを全体の木構造上で表示し，引数の文字の値の流れを局所的に追跡する．その結果，`<2275>`という配列から得られる値であることが分るため，この配列を逆戻りに検索する．

段階 6　`<2275>`の要素の代入のトレースのイベントが表示されるが，段階 4 と同様に文字

コードの値であるため，変換を行う．その結果，<2275>[17]=" "というトレースのイベントが判明する．

段階 7　配列要素に代入される値の流れを局所的に追跡する．この値はいくつかの呼出し元のメソッドに引数として渡された値であることが，木構造の表示を利用した追跡から容易に判明する．追跡の結果，”AB 12...”のような, 属性に設定された文字列から得られた文字であることが分る．

段階 8　属性の文字列を逆戻りに検索する．

```
new String(<668>, 173, 11, ”UTF8”)
```

という, 配列のスライスによって文字列を生成する呼出しが該当する．

段階 9　byte の配列<668>を逆戻りに検索する．要素に代入される値は文字コードの byte 値であるため，段階 4 や段階 6 と同様の変換を行う．その結果，175 番目の要素に空白が代入されることが判明する．

段階 10　配列<668>の 175 番目の要素の代入を全体のトレースで表示し，局所的に追跡を行う．この代入はメソッド addByte で行われ，メソッド addCharToLexer から呼出される．代入される値は addCharToLexer の引数として渡されたものであり，さらに上位の parseValue で得られた値であることが分る．parseValue の実行をソースコードと対応させ，局所的に追跡する．その結果，ループの内側に以下のような条件分岐があることが分り，これが属性の空白を単一化する原因の箇所であることが分る．

```
//Lexer.java: 3447 行目
if (lastc == ’ ’) {
  continue;
}
```

5.2.2　その他の事例

オープンソースプログラムの多くの欠陥は, 単純な文字列検索や 1 段階の追跡によって発見できる．例えば，文献管理アプリケーション JabRef[⋆1]のあるバグ報告 (ID2801333) は，文献の概要の項目の表示に関する欠陥であるが，内部で使われる項目名である"abstract"について検索することで欠陥の箇所を容易に発見できる．

ビルドシステム ant でのバグ報告 (ID41151) は再現性がある故障であり，特定のプロパティの値が更新されないという内容である．基本的に，5.2.1 節で示した jtidy の事例と同様の追跡が可能だが，以下の点で異なる．

(1)　プロパティのキャッシュの利用により実行されなかったコードに注目する必要がある．トレース上に本来利用される値が現れないため，ソースコードを局所的に分析することで追跡を継続する．

(2)　複雑なオブジェクトの組合わせを理解する必要がある．jtidy の文字の配列のように，比較的単純なデータ構造を扱うプログラムとは異なり，apache-ant はプロパティの値が複数のコレクションオブジェクトの組合わせからなる．具体的には，キャッシュのリストを伴うオブジェクトのリストがあり，さらにそのリストは修正を保護するための UnmodifiableList によって包まれている．
Traceglasses では検索によって段階的にオブジェクトの内部の状態を理解することが可能だが，トレースの式においてあるオブジェクトから到達できる値を表現することができない．

5.2.3　考　　察

事例から Traceglasses は欠陥の発見のためのデータと制御の流れの追跡に有用であり，また以下のことがわかる．

- 局所的な追跡には木構造のトレースの表示が有用である (5.2.1 節の段階 7，段階 10)．一般的なブレークポイントデバッガにあるような，呼出しスタックのリストに比べてデータと制御の流れの追跡の効率が良い．
- 単純な依存する全てのデータの流れの追跡ではなく，利用者の判断から追跡対象が決定する (5.2.1 節の段階 3，段階 4)．
- 値の表現が判断の材料になる (5.2.1 節の段階 4，段階 6，段階 9 および 5.2.2 節の ant の事例)．トレース式上の値の変換であれば，Traceglasses の変換機能は有効に働く．複雑なオブジェクトの組合わせの理解のための表現手法の開発は，今後の課題である．

6. 関 連 研 究

以降はいくつかのデバッグに関する関連研究を紹介する．

6.1　逆再生可能なデバッガ

巻き戻し実行が可能なデバッガである Igor[7)]，Bdb[1)]，Koju らの実装[9)]，GDB7.0[⋆1]などは，ブレークポイントで停止した時点からの逆実行により，実行の巻き戻しが可能である．

⋆1 http://jabref.sourceforge.net/

⋆1 http://www.gnu.org/software/gdb/

これらのデバッガはトレースに基づくデバッガと比べて，実行の効率が良いが，一方で大域的な追跡に対しては現実的でない．主な理由は，逆実行のための計算が多く，時間がかかるためである．

6.2　問合せに基づくデバッグ

問合せに基づくデバッグ[3),4),10),13)]は，デバッグ対象のプログラムに対して利用者が論理式などにより問合せを行うことで，欠陥を発見する手法である．このようなデバッグを行うためのツールでは，問合せのための論理式が複雑になり易く，対話的に利用することが困難である．また，実行中のプログラムに対して問合せを行う実装では，局所的な追跡を問合せで行うため，一般的に記述が増えることで時間がかかる．Traceglasses の検索はトレースの値などの情報を利用することが可能で，利用者が対話的に検索キーを決定し追跡することが容易である．

6.3　宣言的なデバッグ

宣言的なデバッグ (declarative debugging)[15)]は，アルゴリズムによるデバッグ (algorithmic debugging) とも呼ばれ，論理プログラミングのデバッグのために提案された手法である．この手法では，デバッグ対象のプログラムの計算過程を木構造として記録する．計算の木構造はある計算とそれに対する部分的な計算の関係からなり，各計算の入力と結果が記録される．欠陥の追跡には，デバッグを行う開発者が木構造のルートから探索を行い，各計算が正しいかどうかを確認する．開発者は誤った結果を返す計算を親から子へ探索していき，最終的に子の結果がすべて正しく，親の結果が誤っている計算を欠陥として特定する．

近年では宣言的なデバッグを Java のような手続き的な言語に導入する研究が行われている．Caballero ら[2)]は，Java のメソッドを計算の単位とし，それらの呼出し関係から木構造を定義する宣言的なデバッガを開発した．Java のような手続き的な言語での宣言的デバッグは状態を扱う必要があるため，論理型言語や関数型言語でのそれと比較してデバッグの効率が低下すると予想される．また，Java のための宣言的なデバッガである JavaDD[5)]は，高級な論理問合せによってデバッグを進める．JavaDD は問合せの対象に，トレースとして記録された具体的な実行時の値を含めることができる．

Traceglasses と宣言的なデバッグの手法は，なんらかの計算結果を記録し，利用する点で共通しているが，Traceglasses は算術演算などを含めた細かいイベントを記録することで利用者の理解を促進させる．また Traceglasses は，宣言的なデバッガでの木構造の探索や JavaDD のようなトレースの検索だけではなく，計算の過程を実行とは逆順にたどる局所的な追跡を含むので，より対話的なデバッグが可能といえる．

6.4　動的なプログラムスライシング

プログラムスライシング[19),20)]は特定のプログラム行に関連する部分的なプログラムを特定する技法である．動的なプログラムスライシングによって，実行時の値に関連したプログラムを得ることができる．この手法を用いて，開発者は実行結果を直接扱うプログラム行を指定し，自動的に欠陥を含む部分的なプログラムを得ることで，欠陥のデバッグに利用できる．関連したプログラムの抽出はデータや制御の依存関係により計算されるが，単純なスライシング手法では得られる部分プログラムのサイズが大きくなり，効率よくデバッグできるとはいえない．近年では Thin Slicing[16)]のようなスライシング手法の改善が提案されている．Thin Slicing は Java のようなプログラムでのベースポインタ (`this` などのメソッド呼出し対象) に関する依存の追跡を省くことで，得られる部分プログラムのサイズを減少させる．この手法は静的なプログラムスライシングとして提案されているが，動的なプログラムスライシングとして応用することができる．

Traceglasses を使った追跡は，プログラムスライシングとは異なり利用者が対話的にデータや制御の依存関係を追跡する．多くの場合，プログラムの計算が不正かどうかは開発者が判断する必要があり，プログラムスライシングにより欠陥の箇所を自動的に特定することは困難である．一方で，Traceglasses と効率のよいスライシング手法を組合わせることで，より効率の良い追跡が行える可能性がある．

6.5　既存のトレースに基づくデバッガ

ここでは既存のトレースに基づくデバッガの実装についてそれぞれ説明する．

ODB[11)]は Java プログラムの逆戻りのデバッグを行うためのツールである．選択した時点に関連するオブジェクトやローカルの値，コンソール出力，ソースコードの表示が可能である．また，トレースを大域的に検索する式言語を持つ．これらの表示は一度に特定のトレースの時点のみを局所的に表示する．そのため，局所的な追跡にステップ操作が必要となる．

TOD(Trace-Oriented Debugger)[14)]は専用の分散データベースによりスケーラブルなトレースの記録を実現した Java に対する逆戻りデバッガである．表示は大域的なスレッドの壁面図 (thread murals)，局所的なステップ操作が可能なメソッドトレースと選択したトレースに関連したオブジェクトの値，IDE 上のソースコード表示からなる．オブジェクトのフィールドの値に対して，その値を設定したトレースに対するリンクを表示し，大域的な追跡を実現する．大域的な検索手段は，オブジェクトの履歴，値の代入時へのリンク，そして文字列によるため，より限定的であるといえる．TOD の実装は記録と値の検索に関して

効率がよく，さらにデーターベースを分散化することで，より大規模なプログラム実行の記録に対応できる．Traceglasses の現在の実装は，記録時の性能低下は TOD にくらべて 1.5 倍程度遅い．Traceglasses の実装には最適化の余地があるものの，この速度差は記録するイベントの種類の量にも原因がある．Traceglasses では配列の操作や算術演算など，TOD が記録しないより細かい粒度のイベントを記録する．

ETV(Execution Trace Viewer)[17),22)] は言語に依存しないトレースを表示可能なデバッガとして開発された．ETV は呼出し関係によるイベントの木構造表示や，選択したイベントに対するソースコード表示，さらにある変数の値の変化に関する履歴表示やオブジェクトのメソッド呼出しの履歴表示などの特徴を持つ．著者らの見る限り，ETV が記録し表示するトレースのイベントは基本的にメソッドや関数の呼出しに限られるため，トレースの木構造表示上での追跡が困難な場合がある．Traceglasses はより細かい粒度のイベントの記録と表示を行う．

Unstuck[6)] は Squeak Smalltalk での逆戻りデバッガの実装である．選択した時点の，メソッドの文脈によるツリー表示，ローカル変数，メソッドの対象と引数の内容，選択したオブジェクトの履歴のトレースの表示，Smalltalk に基づいた式による検索などの特徴を持つ．オブジェクトの履歴のトレースと検索によって大域的な追跡を実現する． 検索の結果はトレースの集合で，木表示上でハイライトされる．ハイライトは木の表示領域に限定されるため，ある値に関する大域的な操作を理解しにくい．

Compass[12)] は，オブジェクトの状態の履歴を効率よく記録する Object Flow VM を利用した Squeak Smalltalk での実装である．プログラムのメソッド呼出しを大域的に表現する魚眼表示や選択したオブジェクトに関する状態の変化のトレース列の表示を持つ．

Whyline[8)] はトレースとプログラムの解析に基づき出力や状態に関する問い合わせを導出する Java のデバッガである．スレッドやメソッドの呼出し文脈のツリーの表示に加え，GUI の画面の状態の表示などを持つ．

これらのデバッガ実装と比較して Traceglasses は，2.3.2 節で述べた問題点を解決し，トレースの式による検索，大域的および局所的な追跡の組合わせの点でより有用であるといえる．

7. 制限と今後の展望

ここでは現在の Traceglasses の制限と将来的な課題について述べる．

現在の Traceglasses では追跡が困難な種類の欠陥が存在する．現在の Traceglasses は，タイミングなどによらない，再現性のある故障を対象としているため，例えばマルチスレッドに関連するような欠陥の追跡には不向きである．また，欠陥の追跡に利用する結果は数値か文字列による表示に限られ，例えば GUI のようなグラフィックスを伴うプログラムの欠陥の追跡には特化していない．Whyline[8)] などのデバッガは GUI の画面やイベントを記録し表示することができる．

課題の一つとして，スケーラビリティの向上が挙げられる．極端にサイズの大きいトレース (例えば 1000 万件を超える場合) は，現在の Traceglasses では表示や検索に時間がかかると予想される．解決の方向としては，トレースのデータを圧縮して性能を向上するか，記録対象であるテストケースのプログラムを分割するなどの工夫によりトレースのサイズを減らすことが考えられる．

5.2.3 節で述べたように，複雑なオブジェクトの内容の理解するための表示方法を開発することが今後の重要な課題の一つである．実験からは，利用者はコレクションオブジェクトに格納された特定の値を，すばやく確認する必要があることが明らかになった．

8. おわりに

本論文では，再現性のある欠陥のデバッグの際に必要な情報と手順を明らかにし，効率の向上のために，新たな欠陥の追跡手法の提案とデバッガ Traceglasses の実装を行った．再現性のある欠陥のデバッグには，プログラムの実行に対して，データと制御の流れの追跡が必要となる．Traceglasses はプログラムの実行を木構造を持つトレースとして保存し，トレース上で効率よく追跡を行うことを可能にした．

事例に対する適用実験から，Traceglasses が実際に有用である利用場面を示した．その場面においては，性能に関して問題なく利用できることも実験結果から示した．

謝　　辞

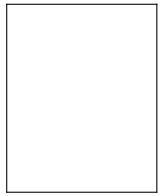
本研究を遂行するにあたり有益な議論とコメントを与えてくださった東京大学 大学院総合文化研究科の増原，玉井両研究室のメンバーならびに，芝浦工業大学 大学院工学研究科の古宮，松浦両研究室のメンバーの皆様に感謝します．

参　考　文　献

1) Bob, B.: Efficient Algorithms for Bidirectional Debugging, *PLDI '00, Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and*

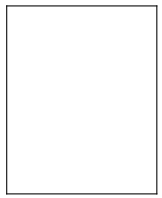
Implementation, New York, NY, USA, ACM, pp.299–310 (2000).
2) Caballero, R., Hermanns, C. and Kuchen, H.: Algorithmic Debugging of Java Programs, *Electron. Notes Theor. Comput. Sci.*, Vol.177, pp.75–89 (2007).
3) Consens, M.P., Hasan, M.Z. and Mendelzon, A.O.: Visualizing and Querying Distributed Event Traces with Hy+, *Applications of Databases, First International Conference, Lecture Notes in Computer Science, Vol. 819*, Springer, pp. 123–141 (1994).
4) Ducassé, M.: Coca: An automated Debugger for C, *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, New York, NY, USA, ACM, pp.504–513 (1999).
5) Girgis, H.Z. and Jayaraman, B.: JavaDD: a Declarative Debugger for Java, Technical report, Department of Computer Science and Engineering, University at Buffalo (2006).
6) Hofer, C., Denker, M. and Ducasse, S.: Design and Implementation of a Backward-In-Time Debugger, *Proceedings of NODE'06, Lecture Notes in Informatics*, Vol.P-88, pp.17–32 (2006).
7) I., F.S. and B., B.C.: IGOR: a system for program debugging via reversible execution, *ACM SIGPLAN Notices*, Vol.24, No.1, pp.112–123 (1989).
8) Ko, A.J. and Myers, B.A.: Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior, *ICSE '08, Proceedings of the 30th International Conference on Software Engineering*, New York, NY, USA, ACM, pp. 301–310 (2008).
9) Koju, T., Takada, S. and Doi, N.: An Efficient and Generic Reversible Debugger using the Virtual Machine based Approach, *VEE '05, Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments*, New York, NY, USA, ACM, pp.79–88 (2005).
10) Lencevicius, R., Hölzle, U. and Singh, A.K.: Query-Based Debugging of Object-Oriented Programs, *OOPSLA '97, Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, ACM, pp.304–317 (1997).
11) Lewis, B.: Debugging Backwards in Time, *AADEBUG'03, Proceedings of the Fifth International Workshop on Automated Debugging* (2003).
12) Lienhard, A., Fierz, J. and Nierstrasz, O.: Flow-Centric, Back-in-Time Debugging, *TOOLS EUROPE 2009, Proceedings of 47th International Conference on Objects, Components, Models and Patterns*, pp.272–288 (2009).
13) Martin, M., Livshits, B. and Lam, M.S.: Finding Application Errors and Security Flaws Using PQL: a Program Query Language, *OOPSLA '05, Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, New York, NY, USA, ACM, pp. 365–383 (2005).
14) Pothier, G., Éric Tanter and Piquer, J.: Scalable Omniscient Debugging, *OOPSLA'07, Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications*, New York, NY, USA, ACM, pp. 535–552 (2007).
15) Shapiro, E.Y.: *Algorithmic Program DeBugging*, MIT Press, Cambridge, MA, USA (1983).
16) Sridharan, M., Fink, S.J. and Bodik, R.: Thin Slicing, *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, New York, NY, USA, ACM, pp.112–122 (2007).
17) Terada, M.: ETV: a Program Trace Player for Students, *ITiCSE '05: Proceedings of the 10th Annual SIGCSE conference on Innovation and Technology in Computer Science Education*, New York, NY, USA, ACM, pp.118–122 (2005).
18) Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagnon, E. and Co, P.: Soot - a Java Optimization Framework, *CASCON'09, Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative Research*, pp.125–135 (1999).
19) Weiser, M.D.: Program slices: formal, psychological, and practical investigations of an automatic program abstraction method, PhD Thesis, University of Michigan, Ann Arbor, MI, USA (1979).
20) Xu, B., Qian, J., Zhang, X., Wu, Z. and Chen, L.: A Brief Survey Of Program Slicing, *ACM SIGSOFT Software Engineering Notes*, Vol.30, No.2, pp.1–36 (2005).
21) Zeller, A.: *Why Programs Fail: A Guide to Systematic Debugging*, Morgan Kaufmann (2005).
22) 柏村俊太朗，寺田　実：実行トレースの特性を活かしたプログラミング支援，情報処理学会夏のプログラミング・シンポジウム「夢をかけるプログラミング〜世代を超えて・夢の再発見〜」報告集，pp.79–84 (2006).

(平成？年？月？日受付)
(平成？年？月？日採録)



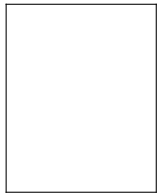
櫻井　孝平

昭和 56 年生．平成 18 年芝浦工業大学大学院工学研究科修士課程修了．平成 21 年東京大学大学院総合文化研究科博士課程修了．同年より芝浦工業大学大学院工学研究科ポスドク研究員．学術博士．アスペクト指向プログラミング言語やソフトウェアテストの研究に従事．ソフトウェア科学会，ACM 各会員．



増原　英彦（正会員）

1970 年生．1992 年東京大学理学部情報科学卒．1994 年同大学大学院理学系研究科情報科学専攻修士課程修了．1995 年同専攻博士課程中退．東京大学大学院総合文化研究科助手・講師を経て 2002 年より助教授 (現准教授)．2001 年から 2002 年までカナダ・ブリティッシュコロンビア大学 Visiting Assistant Professor. 博士 (理学)．先進的なプログラミング言語の設計や実現方式，特にアスペクト指向プログラミング・自己反映計算・部分計算等に興味を持つ．1996 年情報処理学会論文賞受賞．2009 年文部科学大臣表彰若手科学者賞受賞．ACM, 日本ソフトウェア科学会, 情報処理学会各会員．



古宮　誠一（正会員）

昭和 44 年埼玉大学理工学部数学科卒業．昭和 45 年（株）日立製作所入社．昭和 59 年特別認可法人情報処理技術者センター（略称 IPA）に出向し，自動プログラミングシステムをはじめとする各種 CASE ツールの構築技術，ソフトウェア設計方法論とそのメタ理論，CAI および知的 CAI 等の研究に従事．昭和 63 年〜平成 12 年 IPA 技術センター特別研究員．平成 3 年〜平成 9 年 IPA 新ソフトウェア構造化モデル研究本部長付を兼務．平成 5 年徳島大学客員教授．平成 7 年− 22 年 千葉大学情報工学科非常勤講師．平成 9 年より芝浦工業大学客員教授兼同大学大学院非常勤講師．平成 12 年 3 月信州大学博士（工学）．平成 13 年より芝浦工業大学教授．平成 15 年より同大学専門職大学院（MOT）教授を兼務．平成 4・5 年/平成 6・7 年/平成 8・9 年知能ソフトウェア工学研究会幹事/副委員長/委員長．平成 8・9 年電子情報通信学会情報・システムソサエティ運営委員．平成 6 年〜平成 9 年電子情報通信学会論文誌編集委員．平成 10・11 年電子情報通信学会論文誌編集委員．平成 10・11 年電子情報通信学会論文誌編集委員会幹事．現在に至る．