Vol.2009-SE-166 No.9
2009/11/6

対話的に制御可能なトレースに基づくデバッガの開発

藤原一樹[†1]　櫻井孝平[†1]　古宮誠一[†1]

デバッグは，プログラムの故障にいたる不正な出力結果や状態から欠陥を探し出す工程である．デバッグの効率を上げるため，プログラムの実行を記録し，不正な状態から実行とは逆順に欠陥を探す逆戻りデバッガが提案されている．既存の逆戻りデバッガは，プログラム実行の履歴であるトレースの情報をすべて取得する実現方法が一般的だが，それらは一度に扱うトレース情報のサイズが膨大であるという問題があった．本研究では，トレースの量を対話的に制御することで，扱う情報量を抑えた逆戻りデバッガを開発することを目標とする．

The development of the debugger based on trace controllable interactively

Kazuki Fujiwara,[†1] Kouhei Sakurai[†1]
and Seiichi Komiya[†1]

Debugging is a task for finding defects from the program. To achieve efficient debugging, there are back in time debuggers which can record execution of the program and find defects backward from infected states. Existing back in time debuggers generally record all traces of execution history of the program. It is a problem that such the debuggers need to handle large size of the traces. We develop a novel back in time debugger that interactively controls the size of the traces in order to reduce the size.

†1 芝浦工業大学大学院
Graduate School of Shibaura Insutitute of Technology

1. はじめに

デバッグはプログラムコード中の**欠陥** (*defect*) を発見する工程である.[1)] 欠陥を含んだプログラムの実行は, **故障**(*failure*) を引き起こす. 開発者は, 故障にいたるプログラムの実行結果や途中の**不正な状態**(*infection*) から, 原因である欠陥を探す必要があり，一般的にはprint 文の挿入や，ブレークポイントデバッガを利用する．これらの手法やツールは必ずしも故障から原因を発見する作業には向いていない．print 文は必要な情報をえるためにプログラムの修正と再実行を必要とする．ブレークポイントデバッガは基本的にブレークポイントを挿入した時点からプログラムを順に実行するため，前に戻るためには再実行を必要とする．

故障の状態から原因を直接発見するために、プログラムの実行を逆にたどることができる**逆戻りデバッガ**(*back in time debugger*) が提案されている[2)]．逆戻りデバッガを利用したデバッグでは，開発者はプログラムを実行とは逆の順序で，故障にいたる不正な状態をたどっていくことで，故障にいたるプログラムの実行を理解し，最終的に欠陥の発見を行うことができる．不正な状態をたどり実行を理解するために，逆戻りデバッガはプログラムのある実行のある時点について記録された状態と実行されたプログラムコードを得ることができる．これらの情報をもとに不正でない状態から不正な状態へ変化する時点と，そのプログラムコードを発見する．

本研究は，プログラムトレースに基づく新しい逆戻りデバッガの開発を行う．プログラムの実行における，メソッド呼び出し等のすべてのイベント列を記録するような手法をとる逆戻りデバッガは，一般的に記録したトレース情報が膨大になるという特徴を持ち，スケーラビリティは主要な課題の一つとなっている．また，イベント列の記録を行うには，そのためのコードをデバッグ対象プログラムに挿入する方法があるが，それをプログラム全体に対して適用するのは時間がかかる．本研究では，軽量なトレースに基づく逆戻りデバッガを開発することを目標とする．プログラム実行のすべてのイベント列を記録するような手法は取らず，記録するイベント列を利用者が選択し，漸進的に記録するイベント列の範囲を変更しながら，プログラムの再実行を繰り返し行うことでトレース情報のサイズを抑える手法を取る．

本稿は次のような構成になっている．2 節では，一般的なトレースに基づく逆戻りデバッガとその課題を説明する．3 節では，本研究の提案する手法について具体的な事例を用いて述べる．4 節では，本研究で開発した逆戻りデバッガの実装について述べる．5 節では，関

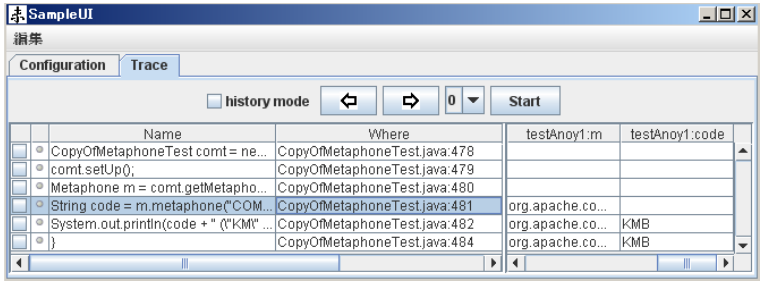

図 1 トレース情報表示画面

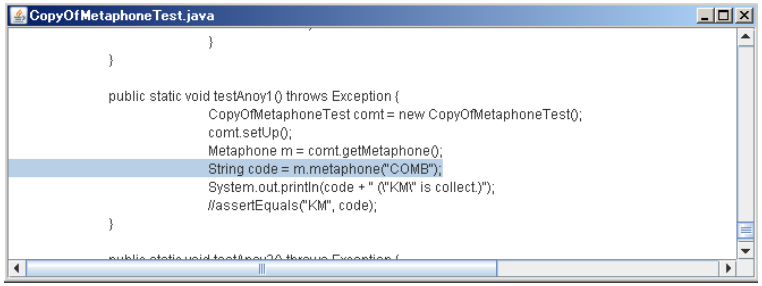

図 2 ソースコード表示画面

連研究について述べる．6 節にまとめと，今後の課題を記す．

2. トレースに基づく逆戻りデバッガ

逆戻りデバッガを実現するために，プログラムの実行をイベント列として記録し，トレース情報としてデバッグに利用する手法がとられる[2)–4)]．本研究で開発するツールは，このようなトレース情報を扱うデバッガである．

この節では，そのようなトレースに基づくデバッガについて説明する．続く 2.1 節では，本研究で開発するツールの，トレースを利用した逆戻りデバッガとしての利用法を説明する．その後，2.2 節では，一般的なトレースに基づく逆戻りデバッガの実装上の課題を述べる．この課題に対する本研究のアプローチは 3 節で提案する．

2.1 デバッグ手順

本研究で開発するデバッガは Java のプログラムに対して，実行の記録を行い，結果のトレース情報を図 1 のような画面で表示する．表の各行がトレース情報としてプログラムの実行の各イベントに実行順に対応する．この表ではイベントに対応するプログラムコードおよびその位置情報 (ソースファイル名と行番号) と，ローカル変数の値を状態として表示する．行番号の部分をクリックすると，図 2 のソースコード表示画面を，該当行をハイライトした状態で表示する．

一般的に逆戻りデバッガを開発者が利用する手順は以下の通りである．

(1) プログラムをデバッガによって実行し，イベントの記録を行う．プログラムの実行は故障を再現するようなテストケースによる入力を前提とする．

(2) トレース情報を確認し，故障の直接の原因となるイベントとその状態に注目する．JUnit[*1]のような単体テストフレームワークを利用したプログラムでは，実行結果の表明文に渡される不正な状態が典型的な候補となる．

(3) 注目する状態の変数に格納されている値の変化を，注目した時点から前の方にさかのぼりながら確認していき，変数に格納されている値が各実行時点における期待値と異なるイベントを探し出す．

(4) (3) で注目したイベントに関連する他の状態が，期待値と異なるかどうかを確認する．そのような状態があれば，その変数について (3) の手順を繰り返す．

例えば，図 3 のプログラムに対してデバッグを行うことを考える．testABC メソッドは getFirst メソッド，getMiddle メソッド，getLast メソッドをテストするテストケースである．getFirst メソッド，getMiddle メソッド，getLast メソッドに引数を与えて呼び出し，その結果を各変数に代入し，全てをつなぎ合わせた文字列が期待値 (文字列”ABC”) と等しくなるかを確認している．getFirst メソッド，getMiddle メソッド，getLast メソッドはそれぞれ入力された文字列の先頭，中間，末尾の文字を返すメソッドである．表の左側は，testABC メソッドを実行した際に，メソッド内で宣言されているすべての変数の各行における期待値が書かれている．この中で，getMiddle メソッドには欠陥が含まれており，中間の文字列ではなく，その一つ後ろの文字列を返してしまうため，テストは失敗する．この欠陥を見つけるためには，まず，テストケースにおけるイベントの記録を行い，表の右側のようなトレース情報を得る．6 行目のイベントに注目すると，assert の対象である abc 変数に期待値と異なる値 (文字列”AIC”) が帰ってきている．トレース情報をさかのぼると，abc の値はひと

*1 http://www.junit.org

```
1  public void testABC() {
2    String a = getFirst(”AHF”);
3    String b = getMiddle(”DBI”);
4    String c = getLast(”GEC”);
5    String abc = a + b + c;
6    assertEquals(”ABC”, abc);
7  }
8
9  public String getMiddle(String str) {
10   return str.substring(
11     str.length() + 1, str.length() + 2);
12 }
```

	期待値				実際の値			
行	a	b	c	abc	a	b	c	
2	A				A			
3	A	B			A	I		
4	A	B	C		A	I	C	
5	A	B	C	ABC	A	I	C	AIC
6	A	B	C	ABC	A	I	C	AIC

図 3 状態の例

つ前のイベントである，5 行目において決定されていることが分かる．この 5 行目において行われている処理には変数 a，変数 b，変数 c が関わっており，表から，b に格納されている値がこの時点における期待値と異なることが分かる．そこで，注目する変数を b に切り替えてトレース情報をさかのぼると，3 行目において期待値と異なる値が代入されていることが分かり，変数が固定値であることから，getMiddle メソッドに故障が含まれることが判明する．このような流れで，プログラムの欠陥を発見する．

2.2 課　　題

プログラムの実行のすべてのイベント列を記録するような手法をとる逆戻りデバッガを，全知のデバッガ (omniscient debugger) と呼ぶ[2]．

全知のデバッガは大抵，プログラムの実行を再現するために観測されるイベント列を状態を含めてすべて記録するため，一般的に記録したトレース情報が膨大になるという特徴を持ち，スケーラビリティは主要な課題の一つとなっている．例えば，全知のデバッガの一つである TOD[3] では，プログラムの実行時の性能低下を抑え，デバッグ時のトレース情報の展開を高速にするために，専用の分散データベースを実装している．また逆戻りデバッガの一つである Whyline[4] に関する実験では 10^5 から 10^7 個のイベント列を扱う複数のプログラムに対して，4MB から 283MB のサイズのトレース情報を扱うことが示されている．

デバッグのためのプログラムの実行は必ずしも時間がかかるわけではなく，たとえデバッグ対象のプログラムのサイズが大きくても，故障を再現するために実行されるプログラムコードと欠陥の発見に必要なトレース情報は，全体の一部にすぎない場合が多い．そのような場合に既存の全知のデバッガは次の 2 つの問題点を持つ．

- 全知のデバッガではイベント列の記録を行うためのコードを挿入するため，プログラムの変換を行う必要があるが，プログラム全体に対して適用するため時間がかかる．
- 欠陥と故障に無関係なトレース情報が記録されることになる．

本研究では，軽量なトレースに基づく逆戻りデバッガを開発することを目標とする．既存の全知のデバッガの手法とは異なり，記録するイベント列を利用者が選択し，漸進的に記録するイベント列の範囲を変更しながら，プログラムの再実行を繰り返し行うことでトレース情報のサイズを抑える．

3. トレースの対話的な制御によるデバッグ

本研究では新しい逆戻りデバッガを開発する．本節では開発したデバッガとその利用手順について実際の事例とともに述べる．

本研究で開発する逆戻りデバッガはプログラムの記録するイベント列を一部に限定し，開発者がそれらを切り替えながら繰り返し記録のための実行を繰り返すことによって，トレース情報のサイズを抑えることを可能にする．

3.1 デバッグ手順

本研究で開発するデバッガは，2.1 節で説明した一般的な逆戻りデバッガの手順のうち (1) と (4) を次のように拡張する．

(1)’ 開発者が選択したプログラムの任意のメソッドをデバッガによって実行し，そのメソッドのプログラムコードによって生じるイベント列の記録を行う．選択するメソッドは典型的なテストケースの実行である．また記録するイベントの種類を図 4 で示す画面で限定することができる．

(4)’ (3) で注目したイベントに関連する他の状態が，期待値と異なるかどうかを確認する．そのような状態があれば，その変数について (3) の手順を繰り返す．このときイベントが他のイベントを含む手続き (メソッドまたはコンストラクタの呼出し) であった場合は，開発者は状態を追跡するためにイベントを選択し，そのイベントで呼出される手続きのプログラムコードによって生じるイベント列を記録する対象に含め，再実行を行う．そして新たに得られたトレースの情報により (3) の手順から繰り返す．

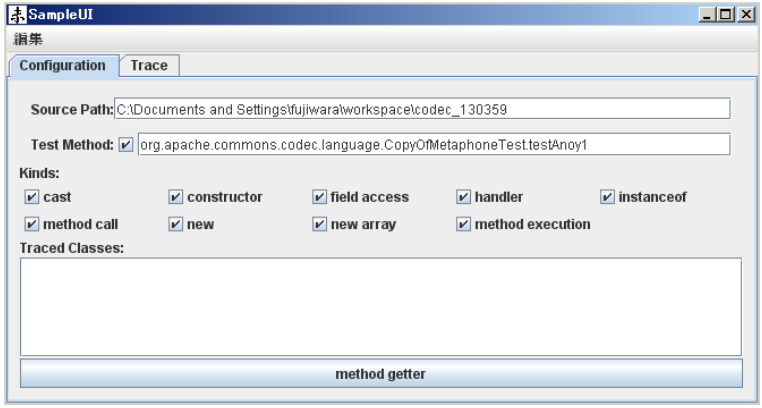

図 4 トレース対象指定画面

3.1.1 事例: Jakarta CODEC

拡張したデバッグの具体的な手順を，Jakarta CODEC[*1]で実際に報告された欠陥を用いて説明する．Jakarta CODEC は String エンコード・デコード用のシンプルなフレームワークであり，ここで扱う欠陥は，文字列の metaphone キーを計算する Metaphone クラスの metaphone メソッドが，特定の入力に対して不正な結果を返すという故障を引き起こすものである．図 5 のコードは報告された故障を再現するテストケースのコードである．このテストケースでは，metaphone メソッドが”COMB”という文字列の入力に対して (482 行目)，結果が期待値である”KM”であるかを確認する (483 行目)．実際には”KMB”が結果となり，テストに失敗する．

本研究で開発する逆戻りデバッガを使ったデバッグの手順は以下のようになる．

(1)’ まず，トレース対象メソッドを testAnoy1 メソッド，追跡対象を assertEquals の対象である code，追跡開始時点を testAnoy1 メソッドの終了時点に設定する．テストケース (図 5) を実行し，testAnoy1 メソッドのすべての変数の値をイベントごとに出力する．

(2) assertEquals によるイベントが期待値との比較であり，失敗していることがわかる (図 6, 484 行目を示す行)．

(3)-1 終了時点から前に変数 (図 6) の値を追跡すると，482 行目で code の値が期待値と異なる値になったことが分かる．482 行目において行われている処理は metaphone メソッド (図 7) の呼び出しである．metaphone メソッドに与えられている引数は不正なものでないため，ここでは，欠陥は metaphone メソッドの記述の中にあることのみが分かる．

(4)’ 次に，トレース対象メソッドを metaphone メソッド，追跡対象を metaphone メソッドの戻り値を格納する変数である code，追跡開始時点を metaphone メソッドの終了時点に設定し，再びテストケースを実行する．

(3)-2 出力された変数の値 (図 8) を追跡すると，148 行目において，code に格納されている値が期待値と異なるものになっていることが分かる．この行において行われている処理は code の末尾への文字の追加である．しかし，本来ならばこの行において実行されるべき処理は，発音を伴わない B を読み飛ばす処理であり break が記述されているはずである．よって，ここが欠陥であると判明する．

上記のように本研究の提案するデバッガでは対象のプログラムを繰り返し実行する必要がある．しかし実際はこのようなテストケースは小さいプログラムであり，実行自体に時間は

```
478 public static void testAnoy1() throws Exception {
479   CopyOfMetaphoneTest comt = new CopyOfMetaphoneTest();
480   comt.setUp();
481   Metaphone m = comt.getMetaphone();
482   String code = m.metaphone(”COMB”);
483   assertEquals(”KM”, code);
484 }
```

図 5 テストケース

Where	testAnoy1:m	testAnoy1:code
CopyOfMetaphoneTest.java:478		
CopyOfMetaphoneTest.java:479		
CopyOfMetaphoneTest.java:480		
CopyOfMetaphoneTest.java:481	org.apache.co...	
CopyOfMetaphoneTest.java:482	org.apache.co...	KMB
CopyOfMetaphoneTest.java:484	org.apache.co...	KMB

図 6 テストケースにおける変数の変化

*1 http://commons.apache.org/codec/

```
136  switch(symb) {
       ...
143    case 'B' :
144      if ((n > 0) && !(n + 1 == wdsz) && (local.charAt(n - 1) == 'M')) {
145        // not MB at end of word
146        code.append(symb);
147      } else {
148        code.append(symb);
149      }
150      mtsz++;
151      break;
```

図 7 metaphone メソッドの一部

Where	metaphone:local	metaphone:code
Metaphone.java:130	COMB	KM
Metaphone.java:132	COMB	KM
Metaphone.java:135	COMB	KM
Metaphone.java:145	COMB	KM
Metaphone.java:148	COMB	KMB
Metaphone.java:150	COMB	KMB

図 8 metaphone メソッドにおける変数の変化

ほとんどかからない．上記のデバッグ手順で得られたイベント列は約 7300 行のソースコードのうち，56 であった．なお，そのうち追跡したイベント数は 18 であった．

4. 実　　装

本節では，開発したデバッガの実装について述べる．本システムは Java を対象としており，実装も Java で行っている．本システムのアーキテクチャを図 9 に示す．全体の流れは次のようになる．

まず，本システムが保持するクラスローダを用いて実行に必要なクラスをロードする．クラスローダの機構に従い，関連するクラスは順次ロードしていく．図 10 に本システムが保持するクラスローダのソースコードにおいて，クラスロードに関わる部分のソースを示す．

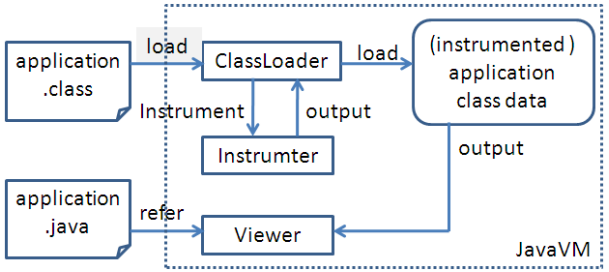

図 9 開発したデバッガのアーキテクチャ

ロードの際，トレース対象に指定されていないものはそのままロードし，トレース対象として指定されているものは instrumenter によってコードの埋め込みが行われる．埋め込むコードの内容は次の 2 つである．一つ目は，トレース対象に指定したメソッドにおいて宣言されている変数の名前と値，位置情報 (ソースファイル名と行番号) を取得するコードである．これは，各行の先頭に埋め込まれる．二つめは，トレース対象に指定したメソッド (コンストラクタ含む) の中で呼び出されているメソッドのシグネチャを取得するコードである．これは，メソッドの呼び出しの直前に埋め込まれる．このコードの埋め込みには Javassist を用いている．コードを埋め込まれたメソッドが実行されると，viewer は種々の情報を受け取る．viewer は種々の情報を受け取ると，該当するソースコード上の箇所を.java ファイルから読み出し，まとめてトレース情報としてユーザに出力する．

クラスの読み込みから書き換えまで，すべてメモリ上で行っているため，.class ファイルが出力されることはない．また，トレース情報もメモリ上でのみ保持しており，ファイル上に出力はしていない．これに関する考察はまとめの節で行う．

4.1 埋め込まれるコードの詳細

埋め込まれるコードの例を図に示す．どちらも図 3 において埋め込まれるコードの例である．実際にはバイトコードに変換して埋め込まれる．

図 3 の 6 行目の先頭に埋め込まれるコードが図 11 である．その行の先頭を実行する時点で宣言されている変数の値を変数の名前をキーとして map に格納する．全ての変数の値を格納した後で，クラス名，ファイル名，行番号，メソッド名とともに viewer に渡される．変数の名前はクラスファイルから取得し，変数がその時点でアクセス可能かどうかはバイトコードオフセットを使って計算する．

図 3 の 11 行目のメソッド呼び出しの直前に埋め込まれるコードの例が図 12 である．get-

```
1  private TraceInstrumenter i = new TraceInstrumenter();
2  protected synchronized Class<?> loadClass(String name, boolean resolve)
3    throws ClassNotFoundException {
4    Class<?> c = findLoadedClass(name);
5    if (c != null)  return c;
6    byte[] data = null;
7    try {
8      data = i.instrument(name);
9    } catch (NotFoundException e) {
10     throw new ClassNotFoundException(e.getMessage());
11   }}
```

図 10 クラスローダの一部

```
trace.LocalVariableMap map = new trace.LocalVariableMap();
map.add("A", a);
map.add("B", b);
map.add("C", c);
map.add("ABC", abc);
TraceResult.v().addTrace(
  new LineTrace().setNames("TestABC", "TestABC.java", 6, testABC, map));
```

図 11 行の先頭に埋め込まれるコードの例

```
TraceResult.v().addTrace(new MethodCallTrace()
  .setNames("java.lang.String", "sustring", Object[int, int], 11));
```

図 12 メソッド呼び出しの直前に埋め込まれるコードの例

First メソッドにおいてはメソッド呼び出しは substring が 1 回，length が 2 回の計 3 つあり，それぞれに対して埋め込みが行われる．図 12 に示しているのは substring の場合である．メソッドを呼び出す直前に埋め込みを行う機構は javassist によって提供される．

5. 関連研究

コードを埋め込む箇所を細かく指定できるツールとして Bugdel[6] が挙げられる．Bugdel はアスペクト指向を利用してデバッグコードを挿入できる Java 用のソフトウェア開発環境である．統合開発環境 Eclipse のプラグインとして実装されており，わかりやすいインターフェースを備えている．同じアスペクト指向システムである AspcetJ/AJDT の持つ機能のほか，新たに line pointcut やローカル変数へのアクセス機能を追加するなど，よりデバッグに特化したシステムとなっている．Bugdel はソースコードに対して，Eclipse を通してコードを埋め込む箇所を指定するが，本システムはトレース情報に対して，出力用の GUI からコードを埋め込む個所を指定する．トレース情報はプログラムの挙動を示すものであり，本システムの指定方法は，結果から原因を探すデバッグに即した，より対話的な方法であると考えられる．

Ko らが開発した Whyline[4] は，プログラムの出力から利用者が理解しやすい質問を生成することによってデバッグを可能にする逆戻りデバッガの一つである．Whyline はプログラムの解析によってユーザーが指定した変数やイベントについて対話的に質問を生成し，不正な状態をさかのぼることができるが，生成可能な質問は限定されているため，どんな場合にも適用できるとはいえない．

6. まとめと今後の課題

本稿では，プログラムの実行を逆にたどることができる逆戻りデバッガを開発した．既存の逆戻りデバッガにおけるトレース情報の量や実行時間の増大対して，対話的にトレース情報を取得することによって解決する方法を示した．トレース情報の取得はトレース対象プログラムを書き換えることで実現している．

本稿で行ったデバッグ法は，各メソッドの処理が正しく行われているかどうかが，実行結果から判断できるという前提のものになっている．そのため，オブジェクトフィールドへの代入や描画などの副作用を持つプログラムのデバッグには対応していない．この点に関しては何らかの対策を講じたいと考えている．

実装に関して言えば，現在は view の出力は各オブジェクトの toString メソッドに依存している．int 等のプリミティブ型と String 型のような単純なデータや，toString メソッドを適切にオーバーライドしているオブジェクトの場合は利用者にわかりやすい値が表示されるが，それ以外のオブジェクト型を view で出力しようとするとハッシュ値が出力されてしま

Vol.2009-SE-166 No.9
2009/11/6

う．これでは null 値でないこと以外が確認できない．解決方法としては，トレース出力画面のほうでオブジェクトを適切な形で表現できる用な機構を用意することがあげられるが，どのような表現が適切であるかについてを検討する必要がある．

また，クラスローダを用いているため，同じようにクラスローダが組み込まれたソフトウェアとの相性が悪い．専用のドライバを用意しているため，ウェブアプリケーションなどに対応できていないという問題もある．これらに関しては，クラスファイルを出力することで，ソフトウェアの種類を選ばずトレース情報が取得できるようにすることを考えている．

参 考 文 献

1) A. Zeller. Why Programs Fail: A Guide to Systematic Debugging. Morgan Kaufmann, 2005.
2) Bil Lewis. Debugging backwards in time. In AADEBUG, 2003.
3) Pothier, Guillaume, Tanter, Éric and Piquer, José, Scalable omniscient debugging, OOPSLA, 2007.
4) Ko, Andrew J. and Myers, Brad A., Debugging reinvented: asking and answering why and why not questions about rogram behavior, ICSE 2008.
5) ”Javassist,”http://www.csg.is.titech.ac.jp/chiba/javassist/
6) Yoshiyuki Usui, Shigeru Chiba, ”Bugdel: An Aspect-Oriented Debugging System,”IEEE Press, pages 790-795, Taipei, aiwan, December 2005.