

平成 1 9 年度修士論文

ユーザ負荷のないUI操作記録・再生型  
デバッグシステム

情報通信工学専攻 情報通信システム学講座

0 6 3 0 0 1 8 柏村 俊太郎

指導教員 寺田 実 准教授

提出日 2 0 0 8 年 1 月 3 0 日

# 目 次

<b>第 1 章</b>	<b>序論</b>	<b>5</b>
1.1	背景	5
1.2	着眼点	5
1.3	目的	6
1.4	本論文の構成	6
<b>第 2 章</b>	<b>関連研究</b>	<b>7</b>
2.1	実行の可視化システムと連動した統合開発環境による GUI ベースプログラムの理 解支援 [1]	7
2.1.1	概要	7
2.1.2	本研究との関連性	8
2.2	インタラクティブプログラムのための操作履歴可視化手法 [3]	9
2.2.1	概要	9
2.2.2	本研究との関連性	9
2.3	Java プログラムの実行履歴に基づくシーケンス図の作成 [4]	11
2.3.1	概要	11
2.3.2	本研究との関連性	11
2.4	JThreadSpy[5]	12
2.4.1	概要	12
2.4.2	本研究との関連性	12
<b>第 3 章</b>	<b>実行トレースシステムの GUI プログラムへの適用</b>	<b>13</b>
3.1	実行トレースシステムについて	13
3.1.1	実行トレースの定義	13
3.1.2	実行トレースシステムの概要	13
3.2	GUI プログラムへ適用するに際する問題点	14
3.3	提案手法	14
3.3.1	既存手法との比較	15
<b>第 4 章</b>	<b>システムの設計</b>	<b>17</b>
4.1	必要な機能	17
4.1.1	ユーザ操作とプログラム動作の関連付け	17
4.1.2	各実行時点における画面情報の提示	17
4.1.3	Java 特有の現象への対応	17
4.2	設計	17
4.2.1	概観	17
4.2.2	pass1: ユーザ操作の記録のための実行	18

4.2.3	pass2: 実行トレース採取のための実行 . . . . .	19
<b>第 5 章</b>	<b>実装</b>	<b>21</b>
5.1	実装環境 . . . . .	21
5.2	pass1 の実装 . . . . .	21
5.2.1	概要 . . . . .	21
5.2.2	EventQueue の置き換え . . . . .	22
5.3	pass2 の実装 . . . . .	23
5.3.1	ユーザ操作の再生 . . . . .	23
5.3.2	イベントハンドラ動作前後のステップカウント記録 . . . . .	23
5.3.3	描画内容の記録 . . . . .	24
5.3.4	画面変化と描画要求との関連付け . . . . .	24
<b>第 6 章</b>	<b>実行トレースの可視化</b>	<b>26</b>
6.1	ETV の概要 . . . . .	26
6.2	GUI プログラムのための情報の可視化 . . . . .	29
6.2.1	画面情報の利用 . . . . .	29
6.2.2	ユーザ操作履歴の利用 . . . . .	32
<b>第 7 章</b>	<b>考察</b>	<b>33</b>
7.1	2pass 方式の影響 . . . . .	33
7.1.1	再現性が得られるもの . . . . .	33
7.1.2	再現性が得られないもの . . . . .	33
7.1.3	再現性を得られるプログラムの条件 . . . . .	33
7.2	パフォーマンス . . . . .	33
7.3	システム利用例 . . . . .	34
<b>第 8 章</b>	<b>結論</b>	<b>37</b>
<b>第 9 章</b>	<b>今後の課題</b>	<b>38</b>
9.1	既存の問題点の解決 . . . . .	38
9.2	適用可能プログラムの拡大 . . . . .	38
	謝辞	39
	参考文献	40

## 目 次

2.1	佐藤らのシステムの概観 . . . . .	7
2.2	中村らのシステムの概観: 1. 操作履歴の概略ストーリーボード、 2. イベント一覧、 3. サムネイル一覧、 4. プレビュー画面 . . . . .	9
2.3	谷口らのシステムによるシーケンス図 . . . . .	11
2.4	JThreadSpy によるダイアグラム: 縦軸がライフライン、 矢印がメソッド呼出し、 スレッドごとに色分けされている . . . . .	12
3.1	本システムを用いたデバッグの流れ . . . . .	15
3.2	各デバッグ手法の比較 . . . . .	16
4.1	Java における GUI イベント処理 . . . . .	18
4.2	イベントの発生と処理の順序 . . . . .	19
5.1	pass1 の概要 . . . . .	21
5.2	pass2 におけるステップカウントの記録 . . . . .	24
5.3	InvocationEvent の記録 . . . . .	25
6.1	ETV の画面構成 . . . . .	27
6.2	Value History View . . . . .	28
6.3	Called Method History View . . . . .	29
6.4	Thumbnail View . . . . .	30
6.5	2 つの移動手段の比較 . . . . .	31
6.6	Event List View . . . . .	32
7.1	本システムの起動構成画面 . . . . .	35

## 表 目 次

7.1 採取情報ごとの pass2 所要時間比較 . . . . .	34
------------------------------------	----

# 第1章 序論

## 1.1 背景

一般に、プログラムのデバッグや理解は困難なものであるが、そのプログラムが GUI ( Graphical User Interface ) を持つもの ( 以降、GUI プログラムと表記 ) である場合、さらに難易度が増すことになる。

その原因の1つとして、GUI プログラムの動作はユーザとのインタラクションに大きく依存するということが挙げられる。これにより、プログラムの動作の解明において有用であるはずのデバッグの有効性が発揮できなくなってしまう。

GUI プログラムはユーザによるマウスやキーボードを用いた GUI コンポーネントへの操作 ( 以降、ユーザ操作と表記 ) を待ち、与えられた操作に従って動作していく。逆に言えば、ユーザが操作を与えない限り操作を待っているプログラムは動かない。

一方、デバッグを用いてプログラムの動作を解析する場合、ステップ実行やブレークポイントなどのデバッグの機能を利用するために、ユーザはデバッグを操作しなければならない。

よって、プログラムを動作させるためにはプログラムの GUI へ操作を与えねばならず、その様子を観察するためにはデバッグを操作しなければならない、という操作同士の干渉が起こってしまう。例えば、GUI コンポーネントをマウスドラッグで移動させつつ、その過程をプログラムが描画する様子をデバッグのステップ実行機能により解析する、などということは不可能である。

## 1.2 着眼点

本研究では、既にいくつか存在するプログラミング支援手法のうち、実行トレースを用いる手法に注目した。

実行トレースとは、プログラム実行の様子の記録のことであり、実行トレースを用いる手法とは、まず初めにプログラムを実行して実行トレースを採取し、それを可視化することでユーザにプログラム実行の詳細を提示する、というものである。この手法では、プログラムの実行と観察をそれぞれ独立して行うことになる。よって、GUI プログラムに適用すれば、プログラム実行時はデバッグ ( デバッグ及び理解の対象となるプログラムの意 ) への操作に、観察時はデバッグへの操作に、それぞれ専念することができ、問題となっている操作の干渉を回避することが可能になる。

しかし、実行トレースを用いたデバッグ手法を GUI プログラムに適用すると、実行トレース採取時に発生するオーバーヘッドによるインタラクションの阻害や、プログラムが停止状態になることによる操作の伝達漏れといった問題が発生する。

### 1.3 目的

本研究では、GUI プログラムのデバッグ及び理解を支援することを目的とし、実行トレースを用いた手法を GUI プログラムに適用する際発生する問題の解決を行う。問題解決のため、実行トレース採取のためのプログラム実行を次の 2pass に分けるという手法を提案し実装する。

1. pass1: ユーザ操作記録のための実行
2. pass2: 実行トレース採取のための実行

実行トレースを用いた手法を適用することで、デバッグ操作とデバugg 操作の干渉を起こさずに GUI プログラム実行の様子を観察することが可能となる。

なお、今回はデバuggとして想定するプログラミング言語を Java とした。

### 1.4 本論文の構成

この章では、序論として、本研究の背景・目的・着目点について述べた。

第2章では、GUI プログラムの理解支援のためのツールや、実行トレースを用いた手法による研究を紹介し、それらと本研究との関連性について述べる。

第3章では、実行トレースを用いた手法を GUI プログラムに適用する際発生する問題点と、それを解決するための提案手法について述べる。

第4章では、本研究で作成するシステムにおいてどのような機能を実装するか、そして、その機能の設計方針について述べる。

第5章では、本システムの実装について述べる。

第6章では、今回新たに採取した、GUI プログラムの動作解析に用いる情報を含んだ実行トレースをどのように可視化しユーザに提示するかについて述べる。

第7章では、本研究で提案した手法によるデバugg対象プログラムの動作への影響や、本システムのパフォーマンス、本システムを用いたデバuggシナリオ例について述べる。

第8章では、本研究によって出た結論を述べる。

第9章では、本システムの今後の課題について述べる。

## 第2章 関連研究

### 2.1 実行の可視化システムと連動した統合開発環境による GUI ベースプログラムの理解支援 [1]

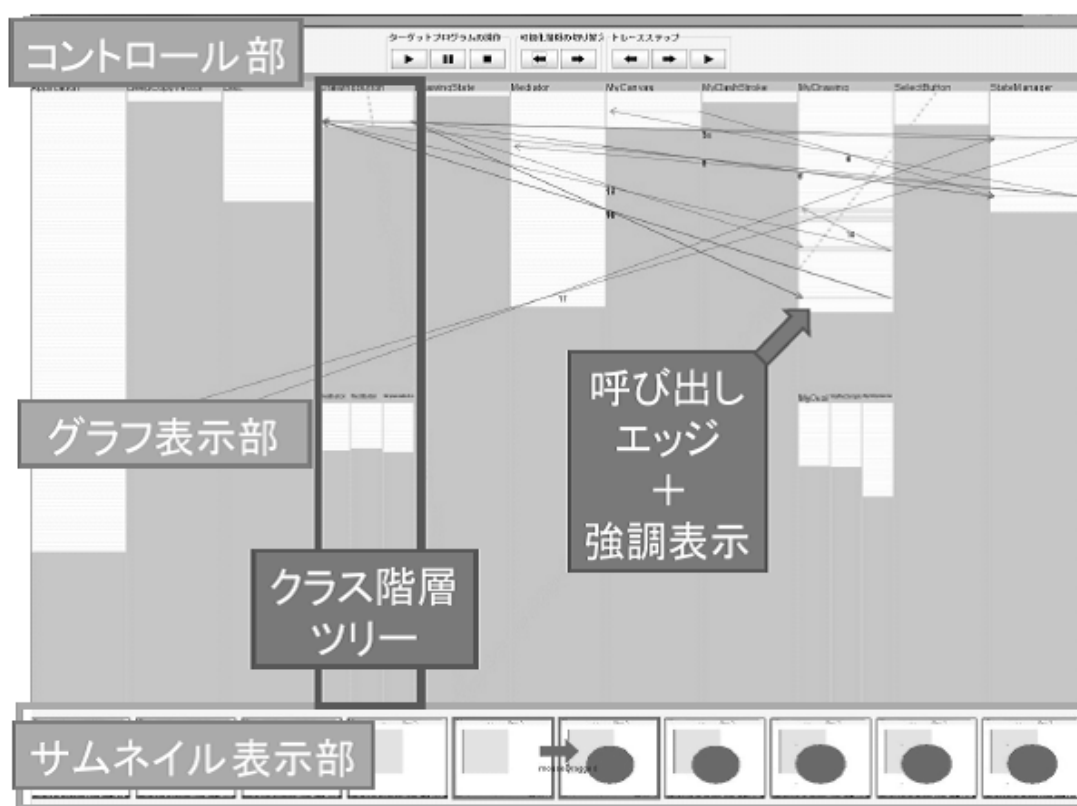


図 2.1: 佐藤らのシステムの概観

#### 2.1.1 概要

統合開発環境 Eclipse [2] 上で動作する、Java の GUI プログラムの理解支援を目的としたシステムである。ユーザによるマウスやキーボードなどを用いたプログラムへの操作と並行してプログラムの解析を行い、操作を行うごとに操作前後の画面情報と実行したソースコードを提示する。プログラムの実行画面と関数呼び出しに重点を置いて可視化を行うことで、ユーザは操作単位でプログラムの構成を把握する。



### 2.1.2 本研究との関連性

Java の GUI プログラムを対象としている点は共通している。

相違点として、このシステムではプログラムの実行と解析を並行して行う点、操作単位でプログラムの挙動を提示する点が挙げられる。本研究では、プログラムの実行と実行情報の解析はそれぞれ独立して行う。また、プログラムの挙動はステップ実行を単位として解析することで、より詳細にプログラム実行の様子を観察する。

## 2.2 インタラクティブプログラムのための操作履歴可視化手法 [3]

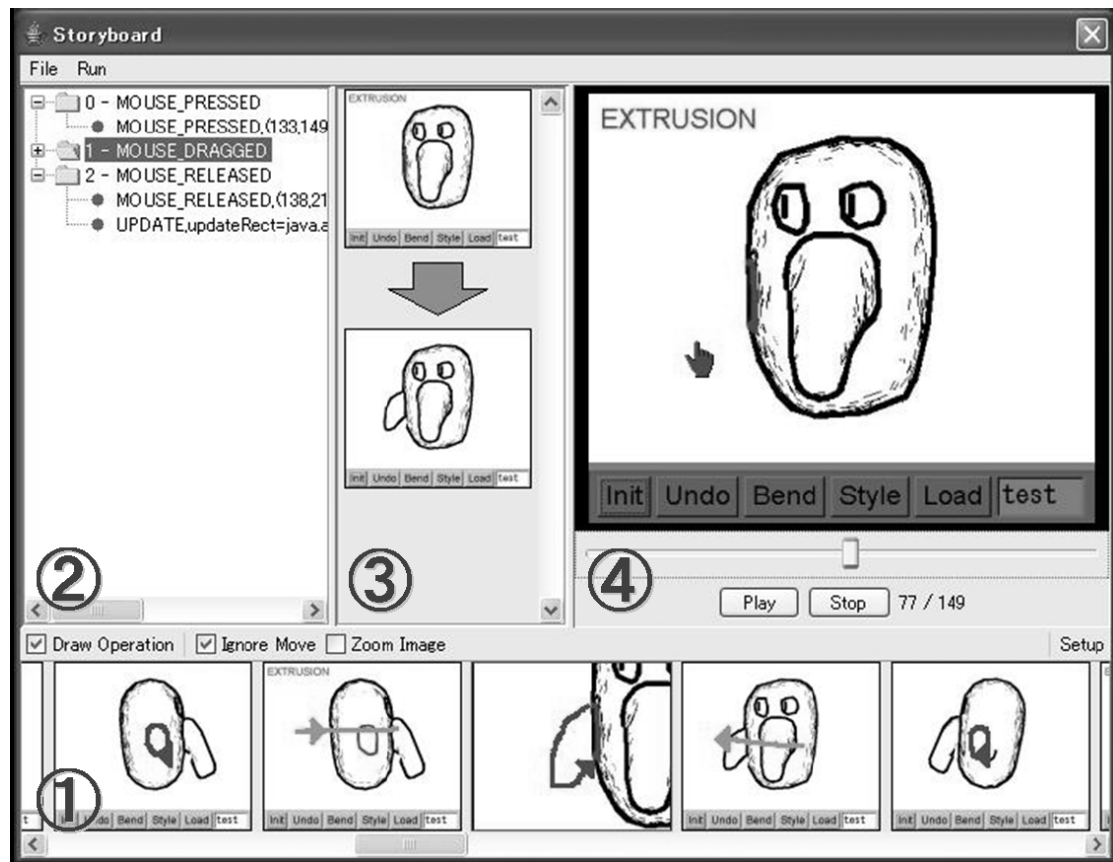


図 2.2: 中村らのシステムの概観: 1. 操作履歴の概略ストーリーボード、2. イベント一覧、3. サムネイル一覧、4. プレビュー画面

### 2.2.1 概要

GUI コンポーネントに対するユーザ操作の履歴を可視化するシステムである。プログラムが描画していた内容をサムネイル表示して提示するが、単に表示するだけでなくユーザが行った操作をサムネイル上に図示するという手法を用いている。これによってスケッチのような動きのある一連の操作を静止画として表現し、ユーザは操作履歴を理解する。また、このシステムではプログラムの実行状態を各サムネイルが表す時点の状態に戻す機能を提供している。

### 2.2.2 本研究との関連性

プログラムが描画していた内容を画像として提示するという点、Java の GUI プログラムを対象としている点は共通している。

相違点として、このシステムでは提示する情報を GUI に関わるもののみに絞っている点、採取した画面情報に操作情報を注釈付けている点が挙げられる。本研究では、変数値などのプログラム実行状態も採取・可視化する。

本研究では画面のどの部分が描き換わったのかという情報を画像に付加しているが、この研究のようにユーザが画面上でどのような操作を行ったのかという情報を加えることで、よりプログラム実行の様子を理解しやすくなることも考えられる。

## 2.3 Java プログラムの実行履歴に基づくシーケンス図の作成 [4]

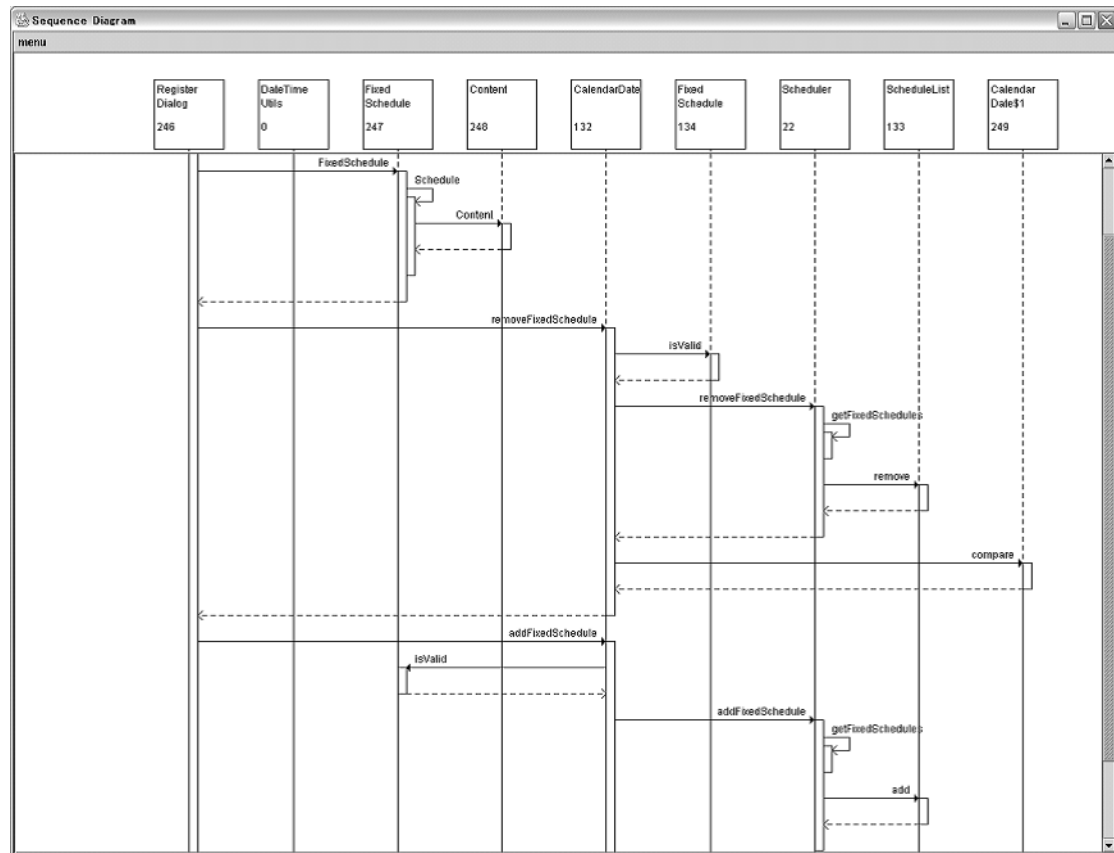


図 2.3: 谷口らのシステムによるシーケンス図

### 2.3.1 概要

Java の実行記録を採取し、UML シーケンス図として提示するシステムである。採取するのは、メソッドの開始と終了に関わる情報に絞っている。さらに、全てのメソッド呼び出しをシーケンス図に示そうとすると巨大な図になってしまうため、繰り返し部分をまとめる等の実行記録圧縮の試みを行っている。

### 2.3.2 本研究との関連性

プログラムを一度動作させ、その様子を採取し解析するという方針は本研究と共通している。しかし、このシステムでは採取する情報をメソッド呼び出しに限定し、プログラムの動作をマクロ視点で把握するための手法を提案しているが、本研究は採取可能な情報は全て採取して解析に利用するという立場を採っており、対照的な位置にあるシステムと言える。

## 2.4 JThreadSpy[5]

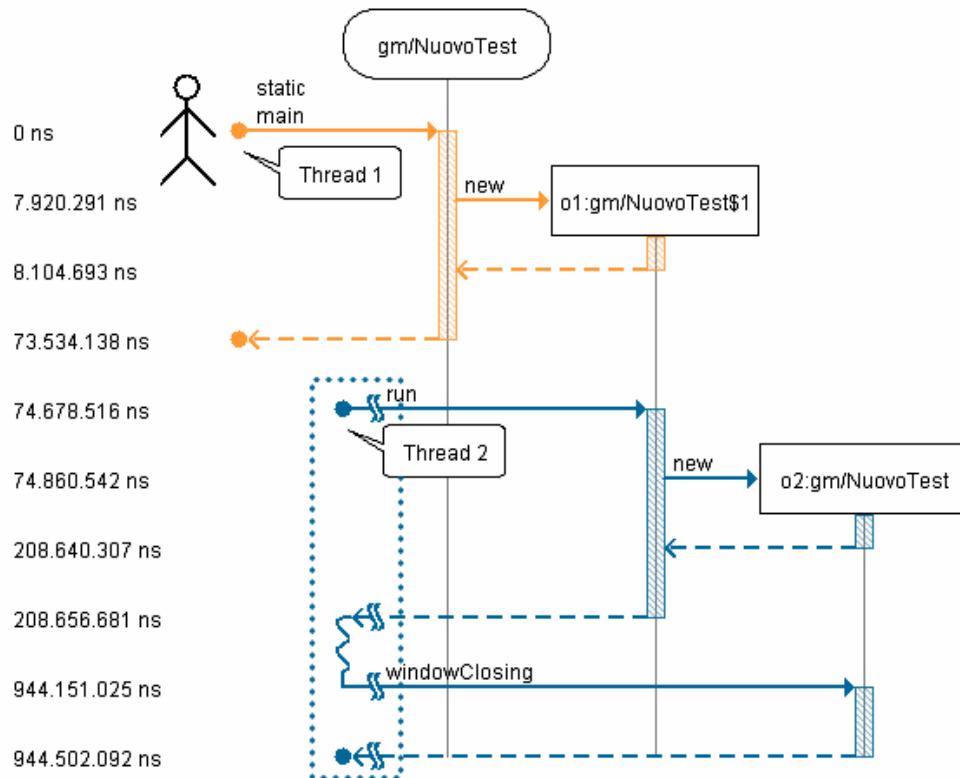


図 2.4: JThreadSpy によるダイアグラム: 縦軸がライフライン、矢印がメソッド呼出し、スレッドごとに色分けされている

### 2.4.1 概要

Java のスレッドに関する実行情報を記録し、各スレッドの動作の様子をダイアグラムとして可視化するというシステムである。スレッドの挙動の提示に特化しており、採取情報もスレッドのモニタの受け渡しやメソッド呼び出し等、マルチスレッドプログラムで要となる項目に絞っている。

### 2.4.2 本研究との関連性

このシステムでも、一度プログラムを実行させ、その様子を記録するという方針を採っている。記録・可視化する情報をスレッド関連に絞っているという点で本研究と異なる。また、このシステムや前述した谷口らのシステムでは、採取した情報を図示するという形でユーザーに提供しているが、本研究ではそのようなデータの加工は行わない。

## 第3章 実行トレースシステムの GUI プログラムへの適用

### 3.1 実行トレースシステムについて

#### 3.1.1 実行トレースの定義

実行トレースとは、プログラムがどのように実行されたのかという記録のことである。これには、プログラム実行中の全実行時点における次のような情報が含まれている。

- 定義されている変数とその値
- メソッド呼出関係
- ソースコード位置
- 実行スレッド

#### 3.1.2 実行トレースシステムの概要

実行トレースを用いる手法では、次の手順でプログラム実行の様子を提示する。

1. プログラムを実行し、実行トレース採取
2. 実行トレースから情報を抽出、可視化

一般に、デバッガの提供する機能はプログラムを実行開始時点から動作させていき、その過程で各実行時点の様子を観察するというものである。そのため、実行時点の移動方向は順方向に限られており、通過してしまった時点へ戻るためには再度プログラムを実行しなおさねばならない。

それに対し、実行トレースシステムでは、プログラムの挙動を観察する時には既にプログラム実行は終了しており、用いるのはソースコードなどの静的な情報と、記録した実行トレースのみである。このことから、次のような特徴を持っている。

実行時点のランダムアクセス その場で実行しているわけではなく、あくまで記録を用いるため、実行順序に囚われることなく任意の実行時点へ移動することができる

履歴情報の活用 各変数の値が、プログラム開始から終了までの間に、いつどのように変わっていったかなどの情報を抽出し利用することができる

これらを組み合わせることで、例えば“ある変数がどのような値を取っていたのかを一覧表示し、ある値に変化した実行時点へ移動する”といった実行時点制御を行うことができる [6]。

## 3.2 GUI プログラムへ適用するに際する問題点

実行トレース採取時には、デバグの実行を制御し、以下の手順をデバグ実行終了まで繰り返す。

1. 1 ステップ実行する
2. 実行を停止する
3. 実行状態をファイルに記録する

このように、実行トレース採取の際にはデバグ実行の停止と再開を何度も繰り返すことになるため、プログラムを通常実行するときと比べると実行速度が大幅に低下するが、特に GUI プログラムを対象とする場合、次のような固有の問題が発生する。

### 応答速度の低下

実行速度が低下するということは、ユーザ操作に対するプログラムの応答速度の低下も招くということである。プログラムにおいて、応答速度が低いというのは忌避される要素の 1 つであり、そのようなプログラムをデバグが終了するまで何度も操作するというのはユーザへの大きな負担になってしまう。

### 操作の伝達漏れ

トレース採取時はプログラムがたびたび停止状態になるが、その間にユーザが行った操作はプログラムに伝達されない。ゆえに、例えば手書きによる自由曲線の描画など、操作に連続性のあるプログラムや操作への応答に実時間性が要求されるプログラムには対応不可能になってしまう。

## 3.3 提案手法

実行トレースを用いる手法を GUI プログラムに適用する際に発生する問題点の解決のために、実行トレース採取のためのデバグ実行を、次のような 2pass に分けるという手法を提案する。

pass1 ユーザ操作の記録のための実行

pass2 実行トレース採取のための実行

pass1 で、ユーザは動作を確認したい操作をデバグに与え、システムはどのような操作が行われたのかを記録する。

pass2 は、pass1 終了後にシステムが自動で行い、ユーザは何もする必要がない。pass1 で記録したユーザ操作を再生することでデバグを動作させ、実行トレースの採取を行う。pass2 では当然オーバーヘッドの発生などが起こるが、自動で行うためその影響がユーザに波及することはない。さらに、デバグへの操作もシステムがユーザ操作記録の再生という形で行うため、デバグ停止によるユーザ操作の伝達漏れも防ぐことができる。

提案手法を用いたデバグの流れを図 3.1 に示す。

pass1: recording events

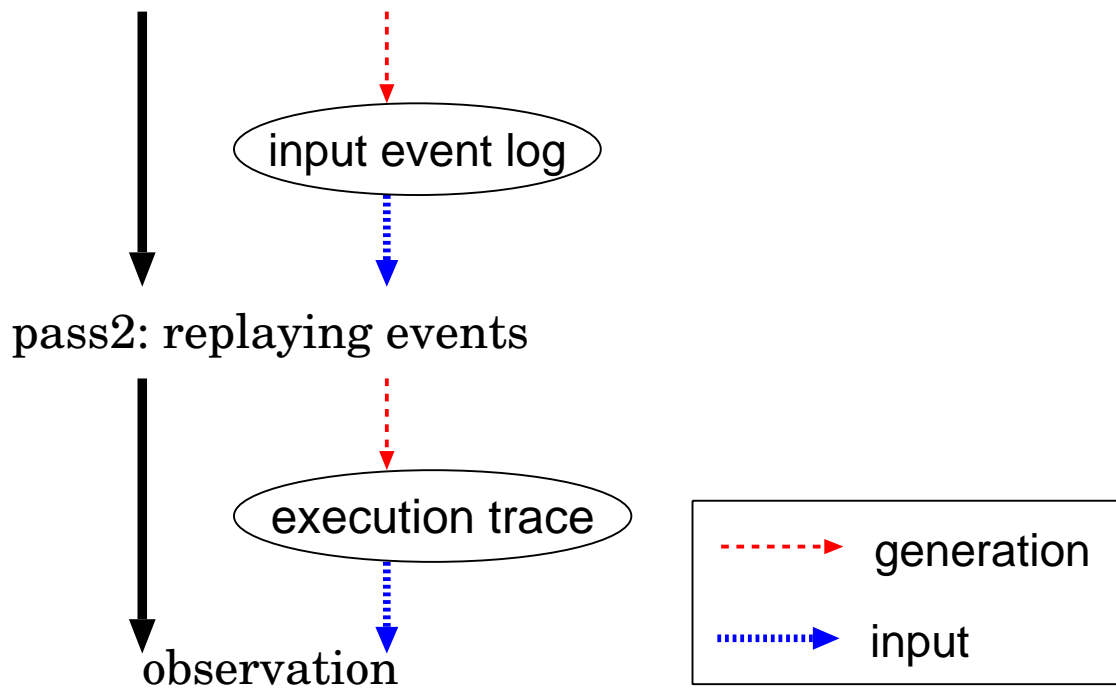


図 3.1: 本システムを用いたデバッグの流れ

### 3.3.1 既存手法との比較

通常のデバッガを用いたデバッグ及び既存の実行トレースを用いたデバッグ手法との比較を図 3.2 に示す。一般のデバッガでは、ユーザが GUI を操作する箇所とデバッガを操作する箇所が重なるため、操作の干渉が起こってしまう。既存の実行トレースを用いた手法では、ユーザが GUI を操作する箇所と実行トレース採取箇所が重なるため、インタラクションの阻害が起こってしまう。本研究で提案する手法では、それぞれを分離することで問題を回避する。



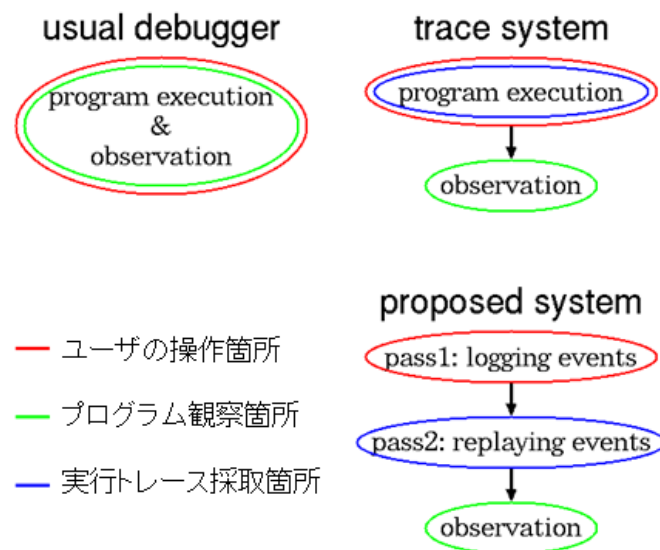


図 3.2: 各デバッグ手法の比較

## 第4章 システムの設計

### 4.1 必要な機能

GUI プログラムに適したデバッガとして備えるべき機能について考える。

#### 4.1.1 ユーザ操作とプログラム動作の関連付け

GUI プログラムは対話的に動作するので、各ユーザ操作に対しプログラムがどのように動作したのかを提示しなければならない。

#### 4.1.2 各実行時点における画面情報の提示

GUI コンポーネントの描画内容は、GUI プログラムを特徴付ける重要な要素であり、デバッグの対象たり得るものである。よって、実際に画面にいつ、どのようなものが描画されていたのかについて提示しなければならない。

#### 4.1.3 Java 特有の現象への対応

Java では、描画要求を出したからといってそれが即座に画面への描画内容として反映されるわけではない。実際にいつ描画されるのかは描画要求を出した時点では不明であり、通常それをプログラムが制御することはない。よって、描画内容が変化した時点で実行中のソースコードを眺めても、それは描画内容の変化とは全く無関係な箇所であるということが頻繁に起こってしまう。しかし、描画内容に誤り箇所があるプログラムをデバッグする場合、画面を誤った状態に描画した原因を特定しなければならない。

## 4.2 設計

### 4.2.1 概観

今日の Java を用いたプログラミングで広く使われている統合開発環境 Eclipse のプラグインという形でシステムを構築する。Eclipse のプラグインとすることで、Eclipse の持つ強力なコーディング支援機能の利用やシステム起動の簡単化といった利便性の向上を図ることができる。さらに、Eclipse では表示する情報の選択や表示レイアウトの変更などを容易に行うことができるため、個々のユーザが目的に合わせてカスタマイズし、画面領域を有効的に使うことができるようになるというメリットも得られる。

システムの起動には、Eclipse で通常にプログラムを起動する場合と同様のインタフェースを提供し、それを用いる。

#### 4.2.2 pass1: ユーザ操作の記録のための実行

##### Java のユーザ操作処理方法

Java の GUI プログラムは、イベント駆動型プログラミングというスタイルで構築する。イベントとはユーザ操作などによって発生するプログラムに対するメッセージのことであり、イベント駆動型プログラミングとは、イベントの発生を待機し、起こったイベントによって対応する処理を行う、というプログラミング手法である。イベントに対応した処理を記述したサブルーチンをイベントハンドラと呼ぶ。

Java では、ユーザが GUI に何らかの操作を行うと、それは次のような手順で処理される。

1. 操作に応じた入力イベントオブジェクトの生成
2. EventQueue に enqueue
3. dequeue 後、対応する GUI コンポーネントに割り当て

Java では、GUI に関わるイベントは EventQueue というクラスのオブジェクトによって管理されている。何らかのイベントが発生すると、そのイベントを表すイベントオブジェクトが生成され、EventQueue に enqueue される。enqueue されたイベントは、enqueue された順序に従って適当なタイミングで dequeue され、GUI コンポーネントに割り当てられる。この時、GUI コンポーネントは受け取ったイベントに応じたイベントハンドラを動作させ、イベントを処理する ( 図 4.1 )。

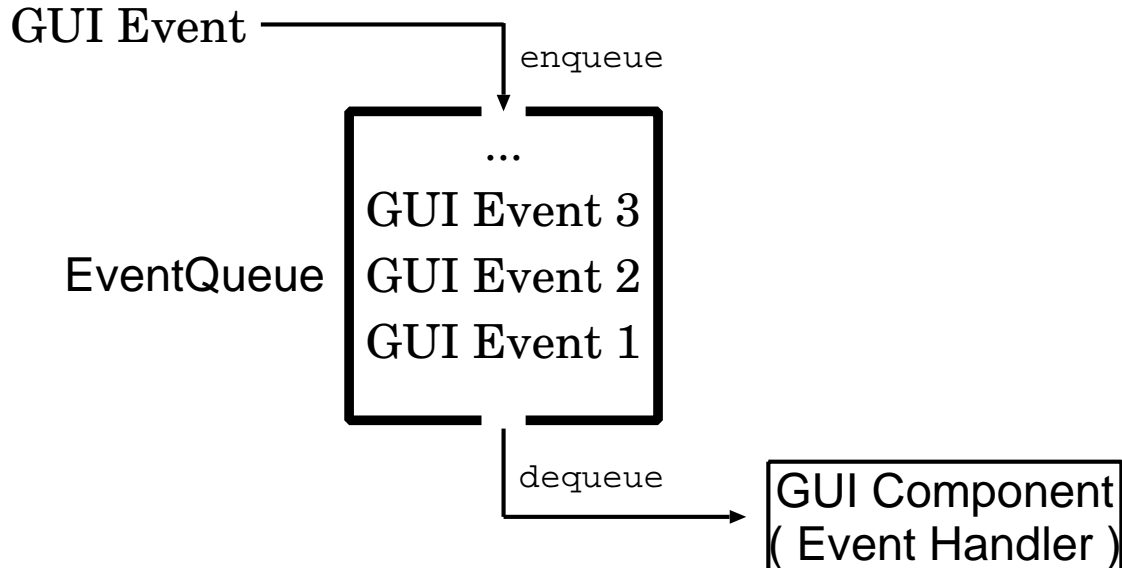


図 4.1: Java における GUI イベント処理

ユーザ操作によるイベントは、InputEvent クラスのサブクラスという形で表現される。例えば、マウスクリックを行った場合には、InputEvent を継承した MouseEvent クラスのオブジェクトが生成され、EventQueue に入ることになる。

## ユーザ操作の記録

Java におけるイベント処理の仕組みから、ユーザがどのような処理を行ったのかを知るためには、EventQueue に入ってくる InputEvent 及びそのサブクラスのオブジェクトを監視すればよいということが分かる。

### 4.2.3 pass2: 実行トレース採取のための実行

#### 操作記録の再生

操作は記録した順序に従い再生する。操作同士の再生間隔は、実時間ではなくイベントの順序関係を基に決定する。入力イベントを処理するイベントハンドラ内でさらにイベントが発生する、ということが多々あるが、このような場合、EventQueue に enqueue される順序は、次のようになる ( 図 4.2 )。

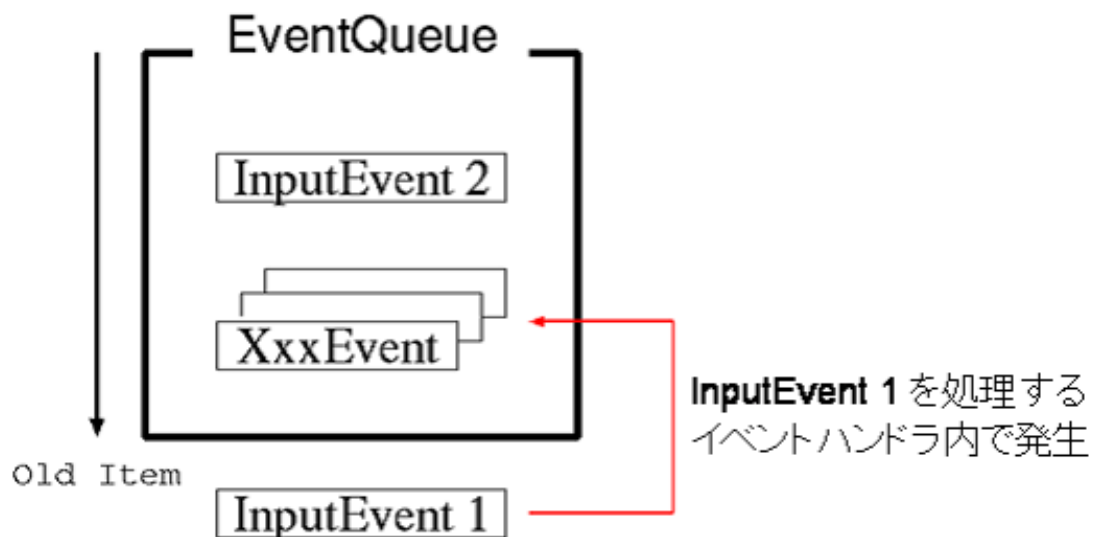


図 4.2: イベントの発生と処理の順序

1. InputEvent 1 が発生し、dispatch される
2. InputEvent 1 を処理するイベントハンドラが動作し、その中で発生したイベントが EventQueue に enqueue される
3. InputEvent 2 が enqueue される

pass2 において、デバッグに pass1 と同じ動作をさせるためには、再生時にもこの順序を再現する必要がある。よって、イベントの再生は次のように行う。

1. 操作 1 を再生する
2. 操作 1 が dequeue され、イベントハンドラが復帰するのを待機する
3. イベントハンドラ復帰を確認したら、操作 2 を再生する

2. のイベントハンドラ復帰時点で、InputEvent 1 を処理するイベントハンドラ内で発生するイベント ( 図 4.2 における XxxEvent の部分が該当 ) の enqueue は完了しており、その後 InputEvent 2 が enqueue されることになる。

#### ユーザ操作とプログラム動作の関連付け

各操作について、イベントハンドラが実行を開始したステップカウントと実行を終了したステップカウントを記録することで、各操作とそれにより実行されたプログラム部分との対応を取る。

#### 描画内容の記録

実行トレース採取のためのステップ実行時に、各ステップ毎に描画内容をフレーム単位の画像ファイルとして保存する。

#### Java 特有の現象への対応

本節の内容は、Java の GUI Toolkit のうちの Swing に対象を限定する。

画面変化が起こったとき、なぜそのように変化したのかを特定するためには、変化した画面情報と、その変化を引き起こした描画要求が出された実行時点との対応を取ればよい。

Painting in AWT and Swing [7] によると、Swing での描画の手順は、次のようになっている。

1. 描画要求が出される
2. 描画要求が EventQueue に enqueue される
3. dequeue 後、対応する GUI コンポーネントに dispatch され、イベントハンドラによる処理の中で実際の描画が起こる

このことから、イベントハンドラが復帰した時点での描画内容変化は、描画要求が出された時点で実行したソースコード行が原因で起こったという関係が成り立つ。よって、ユーザ操作の記録・再生のときと同様、描画要求の EventQueue の出入りを監視し、描画要求が enqueue された時点と、その描画要求が dequeue されイベントハンドラによる処理が終了した時点を記録しておけば、描画内容の変化とその変化を引き起こした描画要求との対応を取ることができるということになる。

## 第5章 実装

### 5.1 実装環境

デバuggiとして想定した言語と同様、本システム自体も Java を用いて実装を行った。実装に用いた Java のバージョンは 1.6.0\_03、Eclipse のバージョンは 3.3 である。

GUI 部分の実装には Eclipse の GUI ツールキットである SWT ( Standard Widget Toolkit ) [8]、SWT をベースにした GUI ツールキット JFace、及び Eclipse プラグイン GEF ( Graphical Editing Framework ) [9] に含まれるプラグイン Draw2D を用いた。

### 5.2 pass1 の実装

#### 5.2.1 概要

EventQueue にイベント記録機能を付加したクラス EventLoggingQueue を作成し、これをデフォルトの EventQueue と置き換えることで実装した ( 図 5.1 )。EventQueue の置き換え方法については後述する。

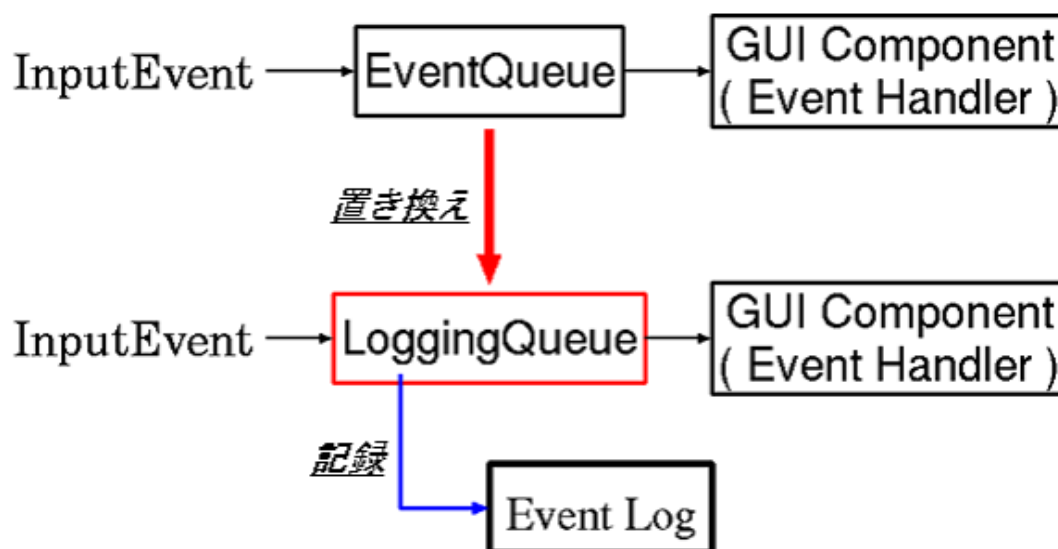


図 5.1: pass1 の概要

EventLoggingQueue は EventQueue のサブクラスである。イベントを GUI コンポーネントに割り当ててイベントハンドラを動作させるメソッド dispatchEvent をオーバーライドすることで、EventQueue が処理するイベントの記録を行う。

EventLoggingQueue の dispatchEvent メソッドの動作は、次のようになる。

1. イベントを記録する
2. スーパークラス ( EventQueue ) の dispatchEvent メソッドを呼び出す

ここで、記録対象オブジェクトのクラスは InputEvent 及びそのサブクラスであり、記録する情報はどのクラスかによって決まる。全ての入力イベントに共通な情報として、次のような情報を記録する。

クラス名 MouseEvent、KeyEvent など

操作の種類 マウスクリック、キープレス など

イベントの発生したフレーム名

さらに、入力イベントがマウス操作を表す MouseEvent の場合、次のような情報を記録する。

- イベント発生ボタン
- イベント発生位置の座標
- クリックカウント
- イベント発生時に押されていた修飾キー ( Shift キーなど )
- イベントがポップアップメニューのトリガであるかどうか

入力イベントが MouseEvent のサブクラスでありマウスホイール操作を表す MouseWheelEvent の場合、MouseEvent で記録したものに加えマウスホイール回転数を記録する。

また、入力イベントがキーボード操作を表す KeyEvent の場合、次のような情報を記録する。

- イベント発生キーの識別コード
- イベント発生キーの文字 ( a キーなら a、shift + a キーなら A など )

他に、フレームの状態変化を示すイベントである WindowEvent のうち、フレームを閉じるというイベントを表現しているもののみ例外的に記録を行う。これは、フレームのタイトルバー部分で行った操作は Java で扱うことができないためである。タイトルバーでの操作例として、フレームを閉じるためのボタンを用いてフレームを閉じ、プログラムを終了させるといった操作が挙げられる。

### 5.2.2 EventQueue の置き換え

EventQueue の置き換えは、イベントの記録漏れを防ぐためにデバグの実行が開始される前に行わねばならない。しかし、デバグの JVM ( Java Virtual Machine: Java のバイトコードをネイティブコードに変換して実行する ) が内部に持っている EventQueue のオブジェクトにアクセスできるのは、当然デバグの JVM が起動してからになる。

そこで、EventQueue を置き換えるためのクラス TraceMain を作成した。TraceMain は、デバグプログラムのメインクラス ( 初めに実行されるメソッド main を持つクラス ) を引数に起動し、次のような処理を行う。

1. EventQueue を置き換える
2. 引数として与えられたクラスの main メソッドを呼び出す

これを用いて、次のような手順でデバッグの起動及び EventQueue の置き換えを行う。

1. TraceMain を用いてデバッグ JVM を起動する
2. TraceMain が EventQueue を置き換える
3. TraceMain が デバッグプログラムの main メソッドを呼び出す
4. デバッグプログラムの実行が開始される

## 5.3 pass2 の実装

### 5.3.1 ユーザ操作の再生

EventQueue を継承したクラス EventReplayQueue を作成し、次のような機能を付加した。

- 不要な入力イベントのブロック
- 入力イベント処理終了の通知

不要な入力イベントとは、本システムが再生した入力イベント以外に入力イベントのことを指す。pass1 と pass2 で発生する入力イベントは共通でなければならないが、pass2 実行時にユーザがマウスを動かすなどの操作を行った場合、pass1 実行時には発生しなかった入力イベントが発生してしまう。そのようなイベントがデバッグに伝達することを防ぎ、再現性を維持する。

また、ユーザ操作を再生するためのクラス EventReliveThread を作成した。このクラスは Thread クラスのサブクラスであり、独自のスレッドで動作し入力イベントを発生させる。これらのクラスを用いて、次のような手順を繰り返すことで記録した全ての入力イベントの再生を行う。

なお、操作の再現には、Java のプログラムから入力イベントを発生させるためのクラス Robot を利用した。

1. EventReliveThread が 1 つ入力イベントを再生し、待機状態になる
2. EventReplayQueue は、入力イベントが enqueue されてくると、それが 1 で再生したものかどうかを判定する。1 で再生したものでない場合、不要なイベントと判断し破棄する。
3. EventReplayQueue は、入力イベントの dequeue 及びイベントハンドラによる処理の終了後、EventReliveThread を再開させる

### 5.3.2 イベントハンドラ動作前後のステップカウント記録

イベントハンドラは dispatchEvent メソッドの処理の中で呼び出される。dispatchEvent メソッドの処理のうち、ユーザプログラムを実行するのはイベントハンドラを呼び出す部分のみであるため、dispatchEvent メソッドの実行開始時点と実行終了時点はイベント処理前後のステップカウントと等しくなる。よって、クラス EventReplayQueue において、dispatchEvent メソッドを次のような処理を行うようオーバーライドすることで実装した ( 図 5.2 )。



1. イベント処理前のステップカウントを記録する
2. スーパークラス ( EventQueue ) の dispatchEvent メソッドを呼び出す
3. イベント処理後のステップカウントを記録する

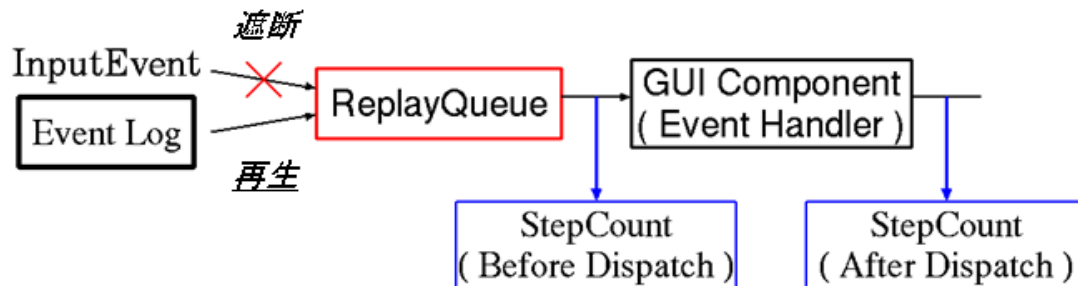


図 5.2: pass2 におけるステップカウントの記録

なお、デバグの起動及び EventQueue の置き換えは、pass1 と全く同様に行う。

### 5.3.3 描画内容の記録

デバグがステップ実行で停止した際に、以下のような手順で取得する。

1. デバグに表示している全フレームの位置・サイズ・名前を要求
2. 受け取った情報を基に、フレームの数だけ画像データを生成
3. 各フレームごと、直前の画像データと比較し、描画内容に変化のあるフレームのみ画像データをファイルに書き出す

ここで、画像の生成と保存を、ユーザ操作の記録のようにデバグ側 JVM で行うのではなく本システム側の JVM で行うのは、動作を制御している JVM で画像の生成と保存のように手間のかかる作業を行うと大きなオーバーヘッドが発生してしまうためである。停止しているデバグと情報のやりとりを行うことでも当然オーバーヘッドは発生するが、それはデバグ内部で画像の生成と保存を行うことによるオーバーヘッドに比べると遥かに小さいものであり、オーバーヘッドの比は 40:1 程度になる。

### 5.3.4 画面変化と描画要求との関連付け

描画要求は、実際にはクラス InvocationEvent のオブジェクトという形で表現されている。よって、InvocationEvent の enqueue 時のステップカウントと、InvocationEvent を処理するイベントハンドラ復帰直後のステップカウントを記録すればよいということになる。

イベントハンドラ復帰のステップカウントは、前述のユーザ操作によるイベントハンドラの場合と同様、dispatchEvent メソッドの復帰のステップカウントを記録すればよい。

EventQueue への enqueue は、クラス EventQueue の postEvent メソッドで行われる。よって、EventReplayQueue において postEvent メソッドをオーバーライドし、次のような処理を行うよう書き換える。

1. enqueue するイベントが InvocationEvent のオブジェクトだった場合、ステップカウントを記録する
2. スーパークラス ( EventQueue ) の postEvent メソッドを呼び出す

InvocationEvent 記録の様子を図 5.3 に示す。

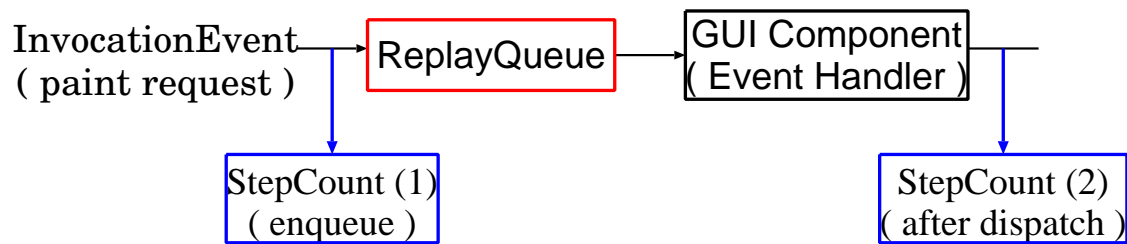


図 5.3: InvocationEvent の記録

## 第6章 実行トレースの可視化

プログラム実行トレース可視化ツール ETV [10] の機能を拡張する形で行う。ETV にはスタンドアロンアプリケーションとしての実装と、Eclipse プラグインとして動作する実装 [11] があるが、今回は後者の方式で実装を行った。

### 6.1 ETV の概要

ETV の画面を図 6.1 に示す。Eclipse の画面はビューという様々な情報を表示するための領域の組み合わせで構成されている。ETV では、実行トレースに含まれる情報を解析・分類し、分類した情報をそれぞれビューに割り当てて表示する。ビューの構成は任意に変更可能であり、不必要なビューを閉じたり、特定のビューを拡大表示したり、ビューの位置を移動させたりといったことができるようになっている。

以降、ETV の提供する各ビューについて説明する。今回追加したのは、⑥ の Thumbnail View と ⑦ の Event List View の 2 つである。この 2 つについては、詳しく後述する。

#### ① Source Code View

**表示** 観察中の実行時点でのソースコードを表示している。背景色が緑に着色されているのが現在実行しようとしている行、灰色の行が現実行時点の所属するスレッドでは実行されることのなかった行である。

**操作** 各行をクリックすることにより、その行を実行する直前の時点へ移動する。また、その行が複数回実行される行だった場合、任意回数目にその行が実行される時点へ移動する、という操作が可能である。

#### ② Control Buttons

**表示** 実行時点の制御をするためのボタンが並んでいる。

**操作** 各ボタンを押すことで、対応する実行時点へ移動する。

#### ③ Thread View

**表示** プログラム中に存在する全てのスレッドと、ステップカウントを表示している。各スレッドの状態を背景色の種類により提示する。例えば、緑色になっているのは現在実行中のスレッドであることを示している。

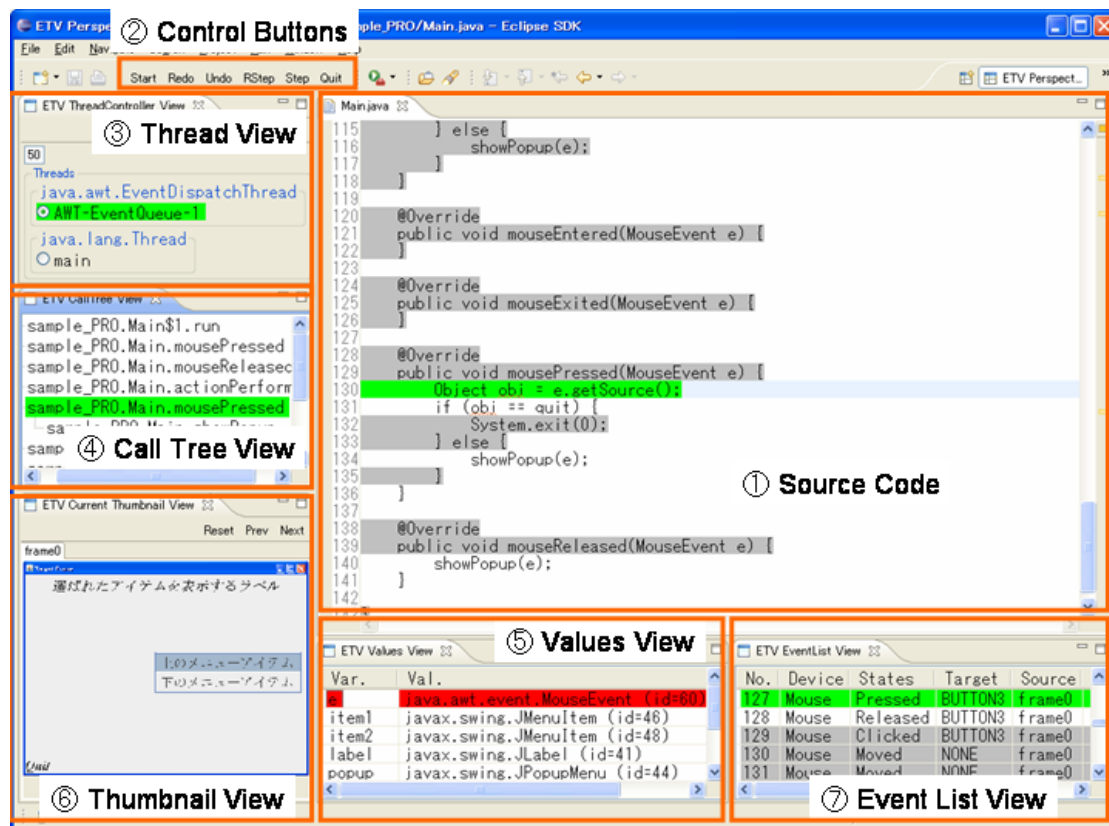


図 6.1: ETV の画面構成

操作 各スレッド名の左にラジオボタンが付いており、選択したスレッドが現実行時点で所持しているモニタと待機しているモニタを表示する。また、任意のステップカウントを入力すると、その時点へ移動する。

#### ④ Call Tree View

表示 メソッド呼び出し関係を木構造で表示している。上下が時間軸、左右が呼び出しの親子関係である。現在実行中の関数は、背景色を緑に着色し強調表示する。

操作 各関数をクリックすると、その関数を呼び出した時点へ移動する。

#### ⑤ Values View

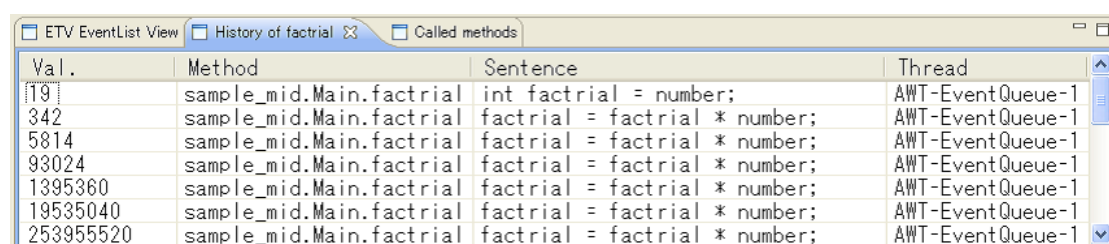
表示 観察中の実行時点で定義されている変数名とその値、及び各変数がローカル変数かインスタンス変数かを表示している。特別な変数として、`this` という名前の変数は現在実行中のメソッドが所属するオブジェクトを表している。現実行時点において変化のあった変数を、背景色を赤に着色することで強調表示する。

操作 各変数をダブルクリックすると、その変数についての Value History View 6.2 を開く。ダブルクリックした変数が `this` だった場合、そのオブジェクトについての Called Method History View 6.3 を開く。

### Value History View

表示 1 つの変数に着目し、その変数がプログラム実行開始から終了までの間にとっていた値の履歴を一覧表示する。観察中の実行時点で着目変数が定義されている場合、一覧中の現在の値の背景色を緑に着色し強調表示する。

操作 一覧中の値をクリックすると、着目変数がその値に変化した時点へ移動する。



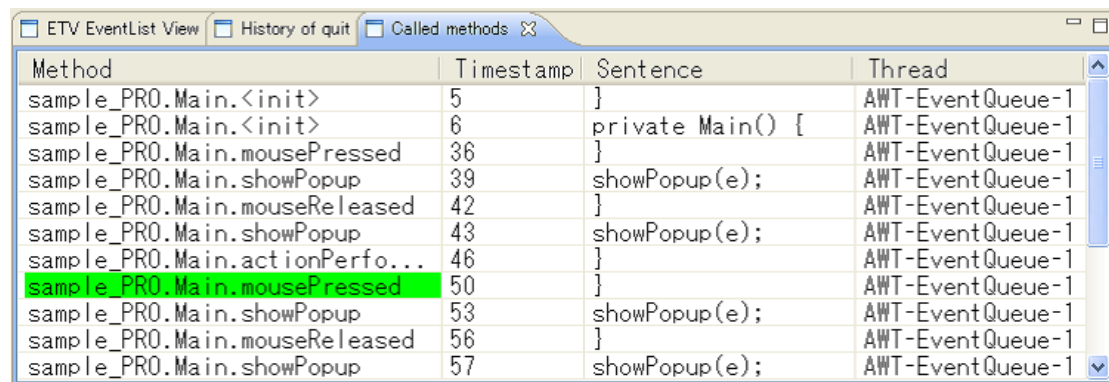
Val.	Method	Sentence	Thread
119	sample_mid.Main.factorial	int factorial = number;	AWT-EventQueue-1
342	sample_mid.Main.factorial	factorial = factorial * number;	AWT-EventQueue-1
5814	sample_mid.Main.factorial	factorial = factorial * number;	AWT-EventQueue-1
93024	sample_mid.Main.factorial	factorial = factorial * number;	AWT-EventQueue-1
1395360	sample_mid.Main.factorial	factorial = factorial * number;	AWT-EventQueue-1
19535040	sample_mid.Main.factorial	factorial = factorial * number;	AWT-EventQueue-1
253955520	sample_mid.Main.factorial	factorial = factorial * number;	AWT-EventQueue-1

図 6.2: Value History View

### Called Method History View

表示 1 つのオブジェクトに着目し、プログラム実行開始から終了までの間に呼ばれたそのオブジェクトのメソッドの履歴を一覧表示する。さらに、各メソッドについて、呼び出し時点のタイムスタンプ、呼び出しを行ったソースコード、実行スレッドを表示する。呼び出し観察中の実行時点が注目オブジェクトのメソッドを実行している場合、一覧中の現在のメソッドの背景色を緑に着色し強調表示する。

操作 一覧中のメソッドをクリックすると、そのメソッドの呼び出し時点へ移動する。



Method	Timestamp	Sentence	Thread
sample_PRO.Main.<init>	5	}	AWT-EventQueue-1
sample_PRO.Main.<init>	6	private Main() {	AWT-EventQueue-1
sample_PRO.Main.mousePressed	36	}	AWT-EventQueue-1
sample_PRO.Main.showPopup	39	showPopup(e);	AWT-EventQueue-1
sample_PRO.Main.mouseReleased	42	}	AWT-EventQueue-1
sample_PRO.Main.showPopup	43	showPopup(e);	AWT-EventQueue-1
sample_PRO.Main.actionPerfo...	46	}	AWT-EventQueue-1
sample_PRO.Main.mousePressed	50	}	AWT-EventQueue-1
sample_PRO.Main.showPopup	53	showPopup(e);	AWT-EventQueue-1
sample_PRO.Main.mouseReleased	56	}	AWT-EventQueue-1
sample_PRO.Main.showPopup	57	showPopup(e);	AWT-EventQueue-1

図 6.3: Called Method History View

## 6.2 GUI プログラムのための情報の可視化

### 6.2.1 画面情報の利用

ETV に、採取した全実行時点での画面情報を提示するための Thumbnail View を追加した (図 6.4)。ここで提示するのは、デバグが観察中の実行時点で画面に表示していた全フレームの画像であり、実行時点の移動と連動して提示する画像も変化する。ユーザが時点移動毎に画像に変化が無いかに注視しなくても済むよう、描画内容に変化があった場合、変化箇所の上に赤い半透明の矩形を描画することで強調表示する。フレームを複数表示していた場合はフレーム名をラベルとしてタブ表示し、どのフレームを表示するかを選択する。画像は、Thumbnail View 領域のサイズ変化に合わせて随時伸縮し、細かい箇所の観察をするために拡大表示することなどが可能となっている。

#### 画面情報を基にした実行時点移動

Values View などと同様に、画面情報の履歴を活用し、次のような時点移動手段を実装した。

前後の画面変化点への移動 表示しているフレームについて、前後の描画内容変化点へ移動する。

操作は、ビュー右上の next ボタンを押すと次の変化点に、prev ボタンを押すと前の変化点へ移動する。

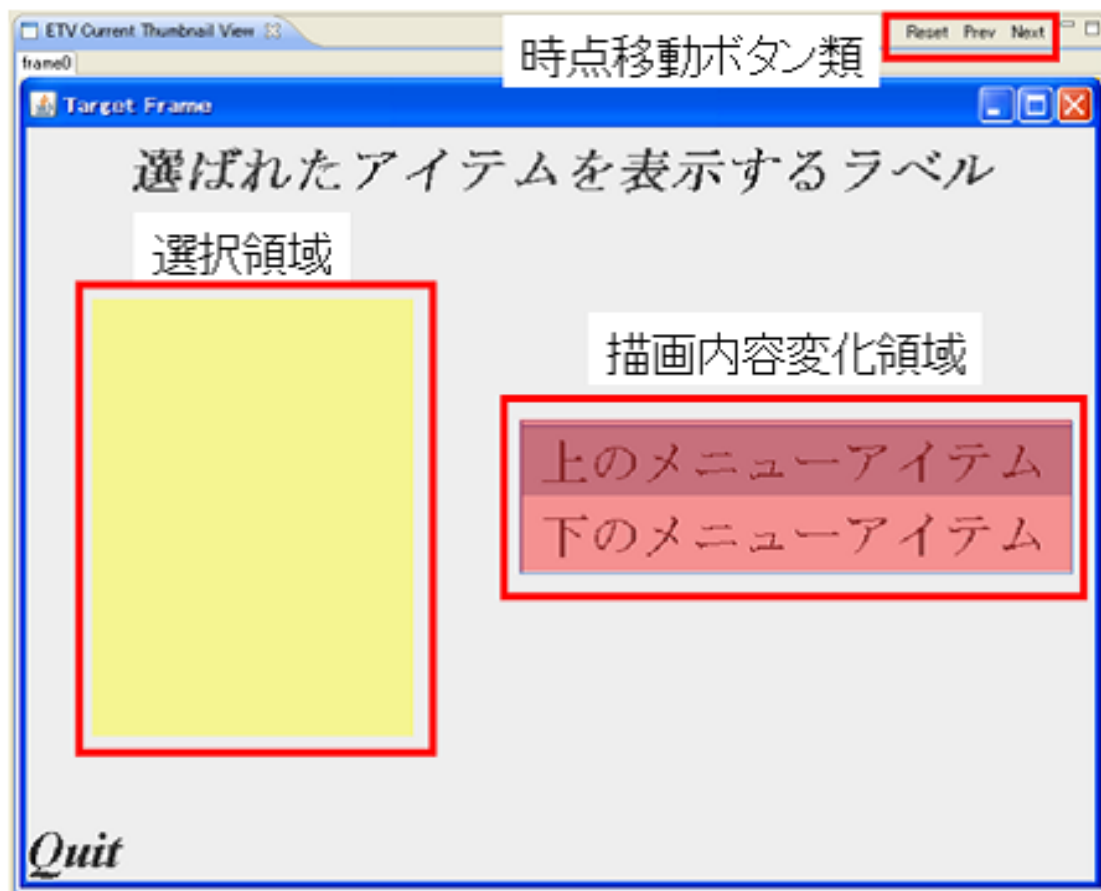


図 6.4: Thumbnail View

任意矩形領域の前後の描画変化点への移動 表示画像上でマウスドラッグすることで矩形領域を選択し ( 図 6.4 の黄色い半透明の矩形領域 )、その領域部分について前後の描画内容変化点へ移動する。操作は、画像上でマウスドラッグを行うと黄色い矩形が描画され選択領域を図示する。その後ビュー右上の next 及び prev ボタンで移動する。また、reset ボタンを押すことで矩形領域の選択を解除できる。

2つの時点移動手段の比較を、図 6.5 に示す。左側が単に次の画面へ遷移していった様子、右側が画面上部のラベル領域を矩形選択してその部分の描画内容変化点へ移動した様子である。左側では初期画面の次にポップアップメニュー表示時点へ、その後ラベル領域の描画変化点へ移動しているのに対し、右側では初期画面の次にラベル領域の描画変化点へ移動している。

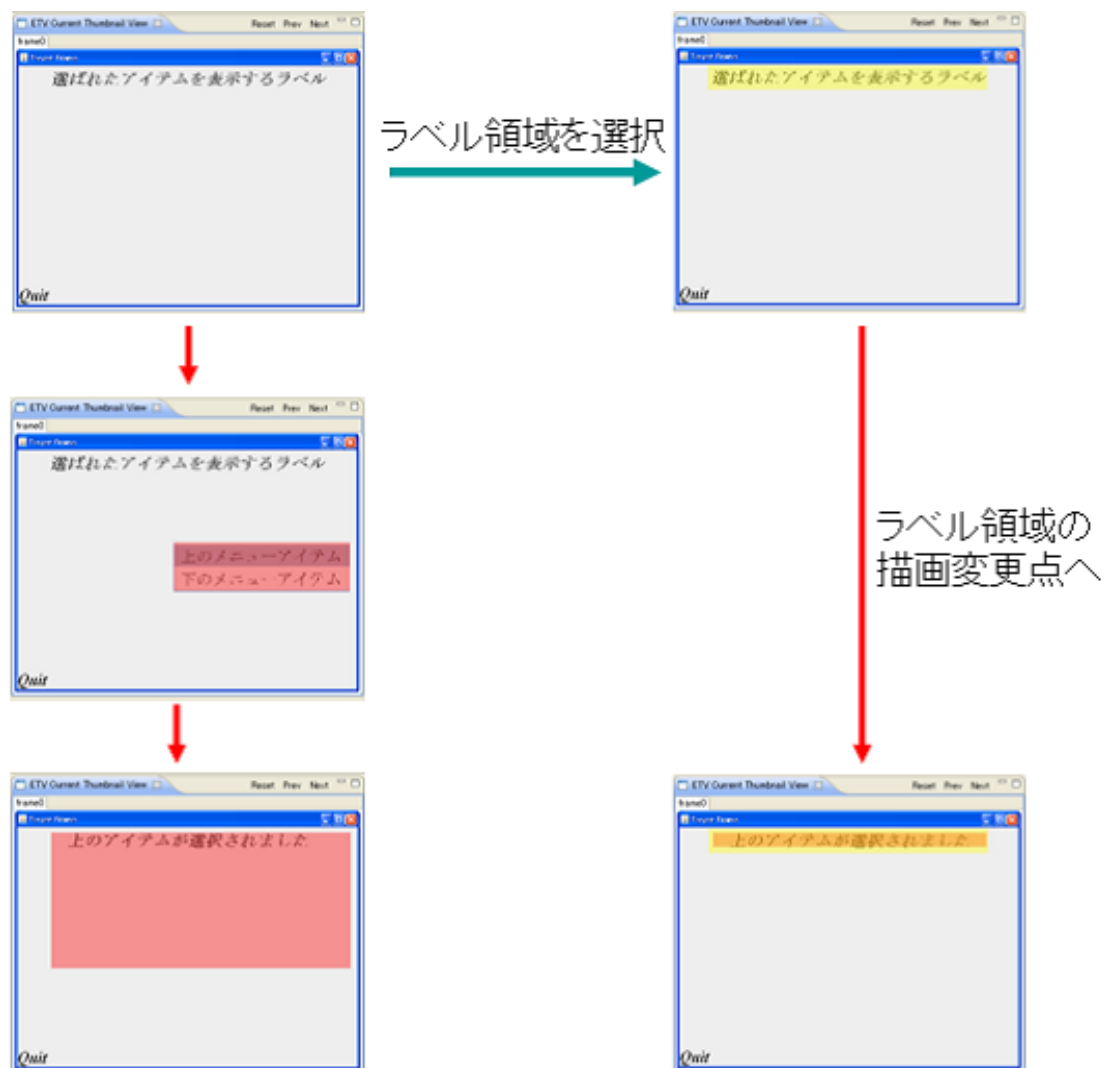


図 6.5: 2つの移動手段の比較

さらに、表示画像をダブルクリックすることで、画面をその画像の画面に描き換える元となった描画要求の出された時点へ移動する。



### 6.2.2 ユーザ操作履歴の利用

ETV に pass1 で記録した全ユーザ操作についての情報を一覧表示する EventListView を追加した ( 図 6.6 )。このビューでは、左から順に、次のような情報を表示する。

イベント番号 何番目に行われた操作なのか

入力デバイス どのデバイスを用いて操作を行ったのか ( マウス、キーボード、等 )

デバイスの状態 どのような操作を行ったのか ( マウスをクリックした、等 )

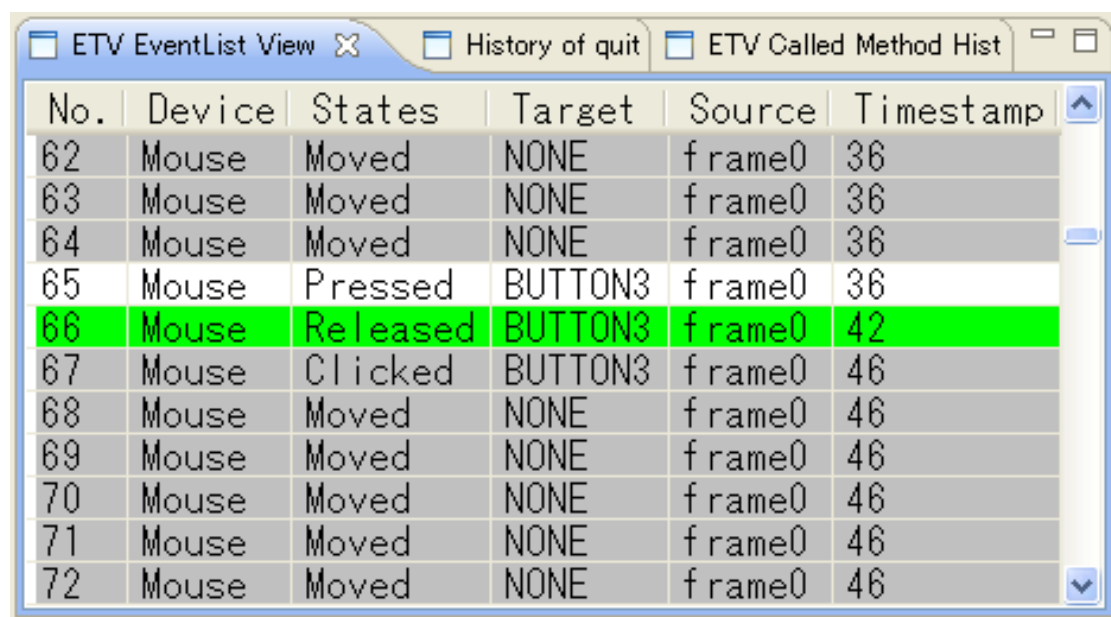
デバイスの部位 デバイスのどの部位を用いて操作を行ったのか ( マウスの右ボタン、キーボードの Enter キー、等 )

操作対象フレーム どのフレームに対し行った操作なのか

タイムスタンプ デバuggiが操作の処理を開始したステップカウント

観察中の実行時点でイベントハンドラによる処理が行われているイベントは、背景色を緑に着色することで強調表示する。また、ユーザが操作を行ったもののイベントハンドラによる処理が行われなかったイベントは、同じく灰色着色しそのことを提示する。

また、各項目とも、最上段の項目名をクリックすることで昇順または降順にソートすることができる。



No.	Device	States	Target	Source	Timestamp
62	Mouse	Moved	NONE	frame0	36
63	Mouse	Moved	NONE	frame0	36
64	Mouse	Moved	NONE	frame0	36
65	Mouse	Pressed	BUTTON3	frame0	36
66	Mouse	Released	BUTTON3	frame0	42
67	Mouse	Clicked	BUTTON3	frame0	46
68	Mouse	Moved	NONE	frame0	46
69	Mouse	Moved	NONE	frame0	46
70	Mouse	Moved	NONE	frame0	46
71	Mouse	Moved	NONE	frame0	46
72	Mouse	Moved	NONE	frame0	46

図 6.6: Event List View

#### ユーザ操作履歴を基にした実行時点移動

一覧中の操作をクリックすると、その操作を処理したイベントハンドラの実行開始時点へ移動する。イベントハンドラによる処理が行われなかったイベントについては、そのイベントが記録されたステップカウントへ移動する。

## 第7章 考察

### 7.1 2pass 方式の影響

今回、実行トレース採取のためのデバグの実行を pass1 と pass2 の2回に分けるということを行った。理想的には、pass1 と pass2 でプログラムの挙動が完全に一致することが望ましい。pass1 と pass2 で動作が変わると、pass1 ではバグが発現したのに pass2 では発現しなかったり、pass1 とは違うバグが pass2 で発現したり、といったことが起こり得るためだ。

ここでは、pass2 が pass1 の再現をできることとできないことについて考える。

#### 7.1.1 再現性が得られるもの

pass2 を動かす入力イベントは、pass1 で記録したユーザ操作を再生することで発生させているため、pass2 で発生する入力イベントの数、種類、順序は再現される。さらに、各入力イベントを処理するイベントハンドラ内で発生するイベントについても、同様に再現される。これにより、イベントを処理するためのスレッドである EventDispatchThread の動作について、再現性を得ることができる。

#### 7.1.2 再現性が得られないもの

マルチスレッドプログラムを始めとして、一般的に複数回の実行で挙動が一致しないプログラムというものが存在する。デバグがそのようなプログラムの場合、pass1 と pass2 でスレッドスケジュール等を再現することができない。

また、時刻や他のホストとの通信等、外部環境に依存して動作するプログラムについても、外部環境の記録等を行っていないため再現することができない。

#### 7.1.3 再現性を得られるプログラムの条件

以上のことから、pass1 と pass2 で再現性を確保できるのは、GUI を初期化した後は単独のスレッドで動作し、外部環境に依存する情報を扱わないようなプログラムということになる。通常のコールバックスタイルで書くような GUI プログラムの場合、概ねこの条件に適合すると考えられる。

### 7.2 パフォーマンス

自由曲線を描画する電子ホワイトボードアプリケーションを作成し、実行トレース採取のための実行である pass2 にどの程度の時間がかかるかを測定した。記録・再生する操作量を次の3パターン用意し、それぞれについて次の3種類の実行を行い所要時間を計測した。

- A 情報の記録を行わない実行、実行制御を伴わない
- B 従来の実行トレースを採取する実行
- C 今回提案した GUI プログラムのための情報を含めた実行トレースを採取する実行

なお、時間計測には、現在の Java で最も精度のよい時刻を返すクラス `System` のメソッド `nanoTime` を用いた。

計測結果を、表 7.1 に示す。各時間の単位は秒である。

表 7.1: 採取情報ごとの pass2 所要時間比較

イベント数	ステップ数	所要時間 (A)	所要時間 (B)	所要時間 (C)	A:B	B:C
137	654	1.51	31.95	96.10	1:21.16	1:3.01
304	1575	2.10	54.55	200.53	1:26.10	1:3.68
440	2243	4.02	75.67	279.64	1:18.82	1:3.70

時間の単位は秒

従来の実行トレースを採取するのに要する時間 (B) と、今回新たに画面情報などを追加した実行トレースを採取するのに要する時間 (C) を比較すると、採取及び記録する情報が増えた分、従来の実行トレース採取と比べ所要時間が増加したことが分かる。しかし、その所要時間の比は、何の情報も採取しない所要時間 (A) と所要時間 (B) との所要時間の比よりも小さいため、実行トレースを採取する手法を選択した場合には、オーバーヘッドの増加よりも採取情報の充実を優先する方が妥当かつ有益であると考えられる。

### 7.3 システム利用例

ここでは、本システムを用いた GUI プログラムのデバッグシナリオ例として、操作に対する画面変化が意図とは違うというバグの誤り箇所を特定する手順を示す。

ユーザは、まず Eclipse 上で通常通りプログラムを作成またはインポートする。次に、プログラムを本システムの提供する起動構成画面を用いて起動する。起動構成画面は、Eclipse に備わっている通常プログラムを実行する際利用するものと同様のインターフェースである (図 7.1)。この画面で、ユーザは観察したいプログラムと与える引数、及び実行トレースを保存するディレクトリを指定する。

起動構成を設定すると、システムは pass1 としてプログラムを起動する。ユーザは、プログラムに動作を観察したい操作を与え、システムはそれを記録する。

pass1 終了後、システムは pass2 としてプログラムを再起動し、記録した操作を再生することでプログラムを動作させつつ実行トレースを採取する。ここではユーザとの対話は何も必要としない。

pass2 終了後、システムは実行トレースを解析し、Eclipse の画面を ETV の画面配置 (Eclipse ではパースペクティブと呼ぶ) に切り替える。

以降は ETV パースペクティブの画面に対し操作していく。まず、画面が意図しない内容に描き換わった時点まで、Thumbnail View の next ボタンを利用して移動する。次にその意図しない内容になるよう描画要求を出した時点へ移動する。そこからは、その時点前後のソースコードに

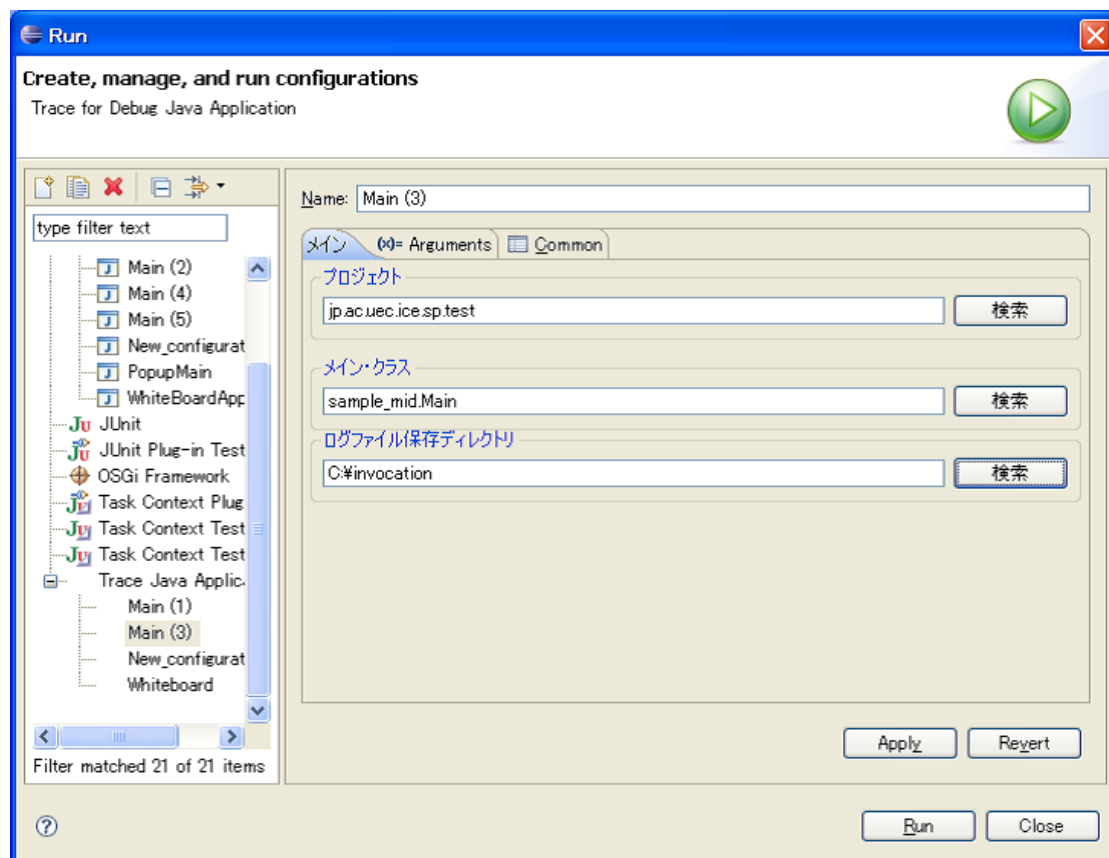


図 7.1: 本システムの起動構成画面

誤りがないか観察する、誤った値になっている変数がある場合その変数の履歴を辿る、などの従来の ETV に備わっている実行時点移動機能を併用して誤り箇所を特定する。

## 第8章 結論

トレースシステムの GUI プログラムへの適用にあたり、問題となっていた

- オーバーヘッドの発生
- イベントの伝達漏れ

の2点を、デバッグ実行を 2pass で行うことで解決し、さらに GUI プログラム特有の情報の記録を併せて行うことで、GUI プログラムに実行トレースを用いたデバッグ手法を適用可能とした。

但し、2回のデバッグ実行で再現性が保障されるのは、GUI コンポーネントの初期化後はイベントスレッド以外のスレッドが動作せず、外部環境による情報を扱わないようなプログラムに限られる。

## 第9章 今後の課題

### 9.1 既存の問題点の解決

今回の実装では、描画内容の保存はステップ実行時に行っているが、描画内容が変化した後にステップ実行が起きないままプログラムが終了した場合、その内容が保存されないという問題がある。ユーザ操作による描画リクエストが出るときには、イベントハンドラが動作するためステップ実行が発生するが、実際に描画された後にステップ実行が起こるとは限らないためである。

また、ユーザ操作の記録及び再生の際、操作対象となるフレームをフレーム名で判別しているため、同じ名前のフレームを複数個生成するプログラムには対応できないという問題がある。

### 9.2 適用可能プログラムの拡大

マルチスレッドプログラムや分散プログラムへの対応を考えている。Java のマルチスレッドプログラムについては、複数回実行時の再現性の問題を扱った研究は既にいくつか行われており [12]、それらの成果を取り入れるなどの方法を検討しつつ解決を目指す。分散プログラムについては、今回 GUI プログラムの実行トレース採取のために導入した 2pass 方式と同様、1pass 目で通信の記録を行い、2pass 目で記録した通信を再現するような仕組みを検討している。

## 謝辞

本研究は電気通信大学大学院情報通信工学専攻寺田研究室において、寺田実准教授の御指導の下で行われました。

寺田実准教授には、研究全般に渡り高い見識に基づいた様々な御助言や御指導を頂きました。心から感謝致します。

東京大学情報基盤センター助教の丸山一貴氏には、研究室輪講や論文執筆等で様々な御助言や御指導を頂きました。深く感謝致します。

博士課程1年の高須賀清隆さん、修士課程2年の鶴原翔夢君、八木原勇太君、修士課程1年の安齋嶺君、学部4年の梅林靖弘君、奥村俊也君、奥村祐気君、中村智行君、平山慧君には、研究についての数々の助言の他、研究室での生活に関わる様々なことでお世話になりました。本当にありがとうございました。

ここには名前を挙げられなかった多くの方々にも御支援・御理解・御協力を得たことを、ここに深く感謝し、併せてお礼申し上げます。



## 参考文献

- [1] 佐藤竜也, 志築文太郎, 田中二郎: “実行の可視化システムと連動した統合開発環境による GUI ベースプログラムの理解支援”, 第 15 回インタラクティブシステムとソフトウェアに関するワークショップ (WISS2007), 日本ソフトウェア科学会, pp.25-30, 2007.
- [2] Eclipse: <http://www.eclipse.org/>
- [3] 中村 利雄, 五十嵐 健夫: “インタラクティブプログラムのための操作履歴視覚化手法”, 第 15 回インタラクティブシステムとソフトウェアに関するワークショップ (WISS2007), 日本ソフトウェア科学会, pp.31-34, 2007.
- [4] 谷口考治, 石尾 隆, 神谷年洋, 楠本真二, 井上克郎: “Java プログラムの実行履歴に基づくシーケンス図の作成”, ソフトウェア工学の基礎ワークショップ 2004 (FOSE 2004), pp. 5-16, 2004.
- [5] Giovanni Malnati, Caterina Maria Cuva, Claudia Barberis: “JThreadSpy: Teaching Multithreading Programming by Analyzing Execution Traces”, Proceedings of the 2007 ACM workshop on Parallel and distributed systems: testing and debugging, pp. 3-13, 2007.
- [6] 柏村俊太郎, 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, 2006.
- [7] Painting in AWT and Swing:  
<http://java.sun.com/products/jfc/tsc/articles/painting/>
- [8] SWT: <http://www.eclipse.org/swt/>.
- [9] GEF: <http://www.eclipse.org/gef/>.
- [10] Minoru Terada: “ETV: a program trace player for students”, Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education (ITiCSE 2005), pp. 118-122, 2005.
- [11] 柏村俊太郎, 寺田実: “実行トレース視覚化システムの Eclipse Plugin としての実装”, 情報処理学会プログラミング・シンポジウム, 2007.
- [12] Jong-Deok Choi, Harini Srinivasan: “Deterministic replay of Java multithreaded applications”, Proceedings of the SIGMETRICS symposium on Parallel and distributed tools SPDT '98, pp. 48-59, 1998.
- [13] H. Lieberman, and C. Fry: “ZStep95: A Reversible, Animated Source Code Stepper”, In J. T. Stasko, J. Domingue, M. H. Brown, and B. A. Price (eds.), Software Visualization, MITPress, pp.277-292, 1998.