特別研究報告

題目

シーケンス図を用いて実行履歴を可視化するデバッグ環境の試作

指導教員
井上 克郎 教授

報告者
淺利 勇太

平成 17 年 2 月 17 日

大阪大学 基礎工学部 情報科学科

平成 16 年度 特別研究報告

シーケンス図を用いて実行履歴を可視化するデバッグ環境の試作

淺利 勇太

内容梗概

プログラムのデバッグ作業では，障害の原因であるバグの位置を特定する作業が最も手間のかかる作業であると指摘されている．オブジェクト指向プログラムにおいてバグ位置を特定するためには，実行時に動的に生成されるオブジェクト群が，どのような実行経路を経て現在に至ったのかを把握することや，不正な状態となったオブジェクトに対してのアクセス履歴を知ることが必要である．このようなオブジェクトの状態を調査する作業については，既有の様々なデバッグ支援ツールでは，十分な支援が得られなかった．本研究では，プログラムの実行履歴をシーケンス図として可視化することにより，デバッグ支援を行うことを目指した．シーケンス図を用いることで，実行履歴における過去のオブジェクト群の動作経路と，各々のオブジェクトへのメソッド呼び出し履歴を理解することが可能となり，バグ位置を特定する作業を支援することができる．シーケンス図の生成には，我々の研究グループで開発してきた実行履歴からのシーケンス図生成ツール Amida を利用した．Amida を統合開発環境 Eclipse にプラグインとして組み込み，デバッガの操作に連動してシーケンス図を生成するデバッグ環境を試作した．試作した環境を用いて実際に Java プログラムを開発し，試作したプラグインの，バグ位置特定における有用性を確認した．

主な用語

デバッグ支援
シーケンス図
動的解析
オブジェクト指向プログラム

目 次

1　まえがき

一般にプログラムのデバッグプロセスは 3 つの作業から構成される．その 3 つとは障害の再現，障害の原因であるバグ位置の特定，そしてバグ修正である．この中でも特に，バグ位置を特定する作業は，プログラムが大規模化，複雑化するにつれて困難なものとなる [1] ．バグ位置の特定のためにデバッガが用いられるが，一般的なデバッガの機能であるブレークポイントやステップ実行では，プログラムが停止した時点での情報しか取得できない．例えば，あるオブジェクトに対してのその時点までのアクセス履歴など，過去の情報は取得できず，それらを取得するためには始めから再度実行しなければならない．また，設計と実際のプログラムの動作とを比較したい場合には，プログラムがどのような実行経路を経たのかを知ることが必要であるが，メソッド呼び出し階層，変数値などの情報だけでは実行経路を再現することはできない．

過去の情報を取得するために，プログラムの逆実行と呼ばれる，実行中のプログラムを以前の状態に戻すという手法の研究も行われている [1] [4] ．しかし，逆実行をどこまで進めるかを開発者が判断するには，やはり，過去にどのような実行経路を経たのかを理解しておく必要があることに変わりはない．

そこで本研究では，デバッグ時に実行履歴をシーケンス図として可視化する手法を提案する．具体的には，デバッガの動作と連動して，実行履歴からのシーケンス図生成を行う．これにより，プログラムがどのような状態を経て現在に至ったかという実行経路や，あるオブジェクトに対する現在までのメソッド呼び出しの履歴が得られ，バグ位置を特定する作業を支援することができる．我々のチームはこれまでに，実行履歴からシーケンス図を生成するシステム Amida を開発してきている [9]．Amida は，オブジェクト指向プログラムにおけるオブジェクト間のメッセージのやり取りなど，プログラムの動作の理解支援を目的としており，プログラムの動的解析から得た実行履歴を基にシーケンス図の作成を行う．プログラムの実行履歴は一般に膨大な量になるが [6][7][8]，繰り返しや再帰構造を圧縮することで，簡潔なシーケンス図を生成することが特徴である．この Amida を統合開発環境 Eclipse[2] にプラグインとして組み込み，デバッガの動作と連動してシーケンス図を生成するデバッグ環境を試作した．そして，このデバッグ環境を，実際の Java プログラム開発に使用し，提案手法の有効性を確認した．

以降，2 章では背景であるデバッグプロセスとその問題点について，3 章では実行履歴からのシーケンス図生成システム Amida について説明する．また 4 章ではデバッグ環境での実行履歴の可視化について，5 章では適用例について，最後に 6 章でまとめについて述べる．

2　背景

本章では，デバッグ作業における問題点と，それに対してシーケンス図がどのように状況を改善するかについて説明する．

2.1　デバッグプロセス

デバッグは，テストによって障害が発見されたときに行われる，障害の除去のための一連の作業である．

デバッグ作業は通常，以下のような手順で進められる．

1. 障害の再現
   デバッグ対象であるプログラムを実行し，その障害がどのような状況で起こるのか，つまり取り除きたい障害を発生させるようなプログラムへの入力や操作を発見する．以降の修正を行った後に同じ入力，操作を行って障害が起こらなければ，その障害は取り除いたとみなすことができる．

2. 障害の原因であるバグの位置の特定
   障害がどのようなもので，その原因がどこに存在し，どこを修正すれば障害が取り除かれるかを特定する．

3. バグ修正
   特定されたバグを修正し，障害が再現しないことを確認する．

これらの作業のうちでも 2 のバグ位置の特定は困難で，時間がかかる作業である [1]．この作業のために，様々な機能を持つデバッガが開発されている．

2.2　デバッガ

統合開発環境 Eclipse[2] における Java デバッガでの機能を例に，主なデバッグ機能を示す．

- ブレークポイント，ウォッチポイント
  ブレークポイントはソースコード上の任意の行に，ウォッチポイントはソースコード上の任意の変数宣言に，それぞれ付加されるマーカーである．デバッガがプログラムの各命令を順番に実行していくとき，ブレークポイントが付加された行の命令を実行する直前，あるいはウォッチポイントが付加された変数への参照や代入の直前で，プログラムの実行を一時停止する．

- ステップ実行
  一時停止状態から，ソースコード上の一時停止位置の次の 1 行の命令を実行し，再び一時停止する．停止中の行の命令にメソッドの呼び出しが含まれている場合，そのメソッドを実装したコードの最初の行で再停止するステップイン，メソッドの呼び出しが終了し呼び出し側の次の 1 行に進むステップオーバーなどがある．

- 停止状態での各種情報の取得
  停止状態では，停止した時点でのスタックトレースやそのスタックフレーム内の各変数の値を取得できる．

開発者はこれらの機能を利用し，デバッグ用に標準出力へ現在の変数値を出力するコードを一時的に追加するなどして，プログラムのソースコード上のどの部分を実行すると障害が発生するかを調査し，バグの位置を特定する．

2.3 現状のデバッガの問題点

上記のデバッガの機能は一般的なものであるが，これらの機能では対処できない問題も存在する．

第一の問題は，プログラムの動作経路が理解しにくいことである．どういう経路を経て現在の状況になったのかということを把握し，想定された動作と比較することでバグ位置を特定することができるが，停止状態でのメソッド，呼び出し階層，変数値などの情報や，ソースコードのみから，複雑なオブジェクト群の動作や関連を理解することは難しい．

第二の問題は，過去の情報は取得できないことである．ブレークポイントやステップ実行などで，一時停止した時点でのメソッドの呼び出しスタックや変数の値は取得することができる．しかし，停止状態においてあるオブジェクトが不正な状態を示していることが分かった場合，次に利用者が知りたいのは，この不正なオブジェクトに対してそれまでにどのようなアクセスがあったのか，ということである．それを知るためには，現状のデバッガの機能では，新たにブレークポイントを設置して，もう一度始めからデバッグ実行を行わなければならない．

過去の情報を取得するために，プログラムの逆実行と呼ばれる，実行中のプログラムを以前の状態に戻すという手法の研究も行われている [1] [4] ．これは，実行が進むたびにプログラムの変数やスタックトレースなどの状態を記録しておき，後で，記録した状態に順番に戻っていくアプローチである．これによって逆順にたどって行けば，任意の時点での情報を得られる．しかし，逆実行をどこまで進めるかを開発者が判断するには，やはり，過去にどのような実行経路を経たのかを理解しておく必要があることに変わりはない．

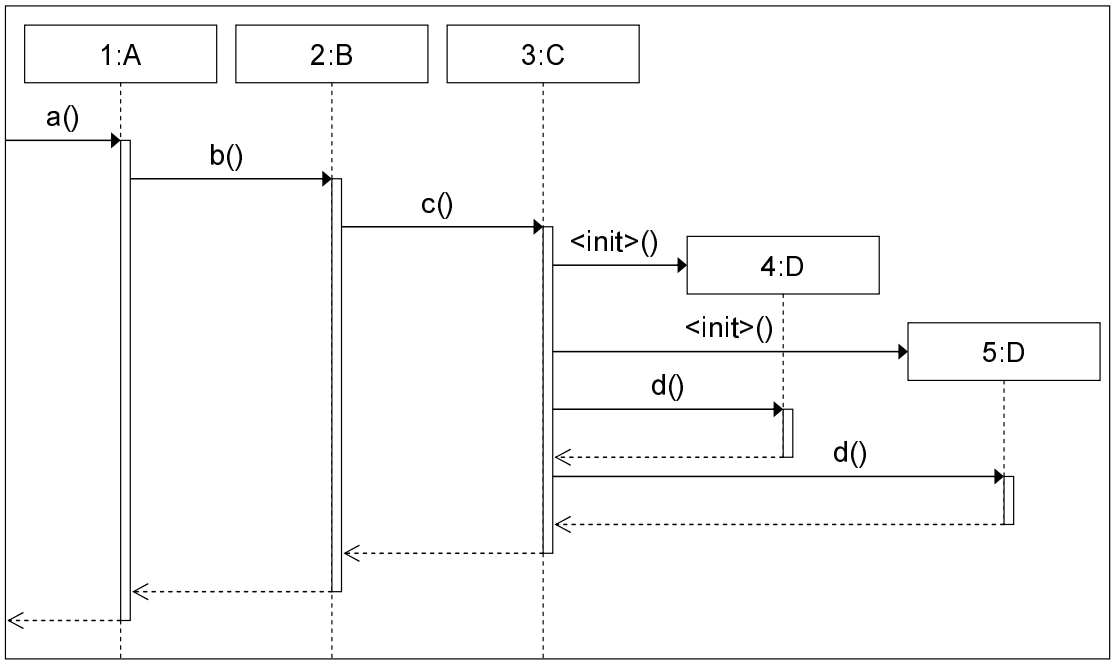

図 1: シーケンス図

本研究では，デバッグ時に実行履歴をシーケンス図として可視化する手法を提案する．他のデバッグ機能と連動して，実行履歴をシーケンス図として可視化することによって，実際のオブジェクト群の動作理解を支援し，また，あるオブジェクトに対するメソッド呼び出し履歴を示す．

我々のチームでは，実行履歴からシーケンス図を生成するシステム Amida[9] を開発してきている．Amida では，膨大な実行履歴の中から繰り返しや再帰構造を圧縮することで，簡潔なシーケンス図を生成することを可能としている．本研究では，Amida をデバッガと連動させることで，簡潔なシーケンス図を用いたデバッグ作業を可能にする．

2.4 UML シーケンス図

シーケンス図 (図 1) とは Unified Modeling Language(UML)[10] で定義されているインタラクション図の 1 つで，オブジェクト間のメソッド呼び出しやオブジェクトの生成などのメッセージ通信を，時系列に沿って示すことができる図である．横軸はオブジェクトの種類を表しており，図 1 のように，図の上部には，図中に記述されるメッセージ通信に関連するオブジェクトが横方向に並べられる．縦軸は時間軸を表しており，下方に行くほど時間が経過していく．各オブジェクトの下部には縦方向に点線が引かれており，これがオブジェクトが生存する区間を示している．さらに，個々のメッセージ通信について，時系列順に，送信元の

オブジェクトから，送信先のオブジェクトに対して矢印を引く．送信されたメッセージがメソッド呼び出しだった場合は，メッセージを受けたオブジェクトは，そのメソッドの実行区間を縦長の長方形で表す．メソッドが終了する時点で，呼び出し元のオブジェクトへ戻り辺の矢印を引く．メッセージがオブジェクトの生成だった場合は，生成されるオブジェクトをその高さに書く．このようにして，オブジェクト間のメッセージの様子を時系列に沿って表現する．

シーケンス図を用いると，メッセージ送受信の関係をたどっていくことにより，システムのどのオブジェクトが動作しているかを知ることができ，システムが現在どのような処理を実行しているか，また以前にどのような処理を実行していたかを理解することができる．また，あるオブジェクトに注目したとき，そのオブジェクトへのアクセスの履歴は，シーケンス図上で生存区間を時間軸の過去の方向へさかのぼっていくことで得られる．

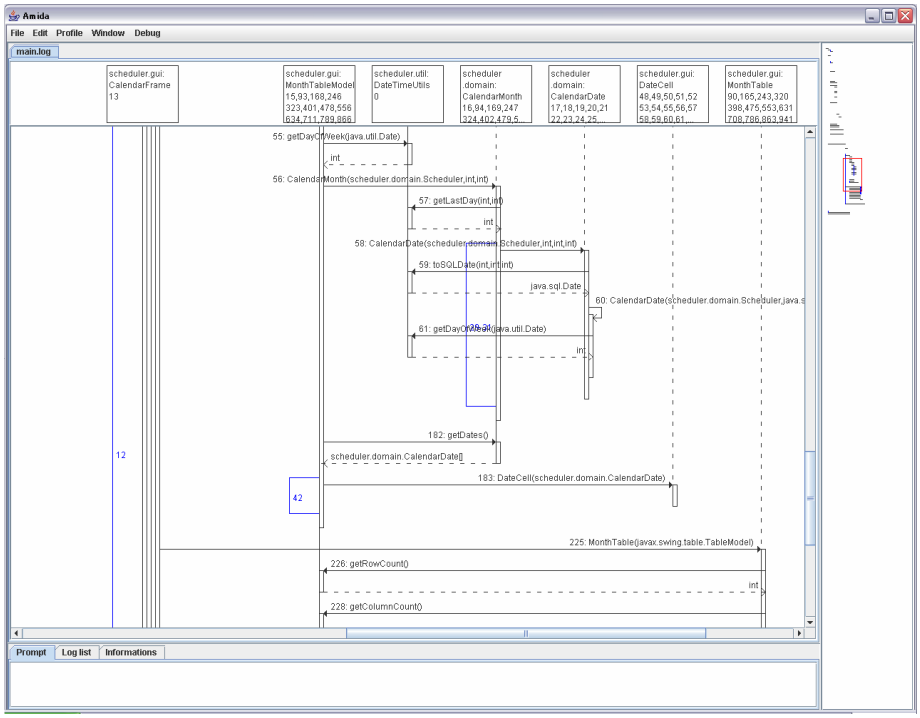

図 2: 実行履歴からのシーケンス図生成 Amida

3　実行履歴からのシーケンス図生成 Amida

我々は，オブジェクト指向プログラムにおけるオブジェクト群の動的な振る舞いを視覚的に表現し，プログラムの理解支援を行うために，プログラムの動的解析から得た実行履歴を基にシーケンス図の生成を行うツール Amida を作成してきている [9].

Amida は Java プログラムを対象とし，プログラムのメソッド呼び出しを実行履歴として取得 · 解析してシーケンス図を生成，GUI にて表示する．また，実行履歴は膨大になるため，繰り返しや再帰の圧縮を行うことで，簡潔な図を生成する．

本章では，この Amida について簡単な説明を行う．

3.1　動的解析による実行履歴の取得

シーケンス図は，オブジェクトの生成とオブジェクト間のメソッド呼び出しについて，時系列に沿って表現する図である．オブジェクトの生成のメッセージとメソッド呼び出しのメッセージは，シーケンス図上では別の形式で表現されるが，本研究で対象としている Java 言語においては，オブジェクトの生成は，コンストラクタの呼び出しとみなせるため，実行履歴上は同様のものとして扱える．以降は，メソッド呼び出しという時は，コンストラクタ

の呼び出しも含めることとし，オブジェクトの生成のメッセージをメソッド呼び出しのメッセージと同種のものとして扱う．

本研究では，対象とするプログラムに対して動的解析を行い，オブジェクト間のメソッド呼び出しに関する情報を実行履歴として記録する．具体的には，個々のメソッド呼び出しについて，メソッド開始時にクラス名，オブジェクト ID，メソッド名，引数の型，返り値の型を記録する．また，メソッド終了時には終了記号を記録する．引数の型を記録するのは，メソッドがオーバーロードされて同名のメソッドが複数存在する場合に，メソッドを特定するためであり，実行時に引数として与えられた値については記録しない．これらの情報を用いることで，実行時に呼び出されたオブジェクトとメソッドを特定し，メソッドの呼び出し構造を再現することが可能となる．

実行履歴を取得するシステムは，対象とするプログラムの実行中に個々のメソッド呼び出しを捉え，実行履歴を保存するファイルに「**返り値の型 クラス名 (オブジェクト ID). メソッド名 (引数の型)**{」，という形式で記録する．static メソッドの呼び出しについては，オブジェクト ID を 0 番とする．閉じ括弧「}」のみの行は，それぞれ対応する開き括弧のメソッドの終了を表している．

実際に取得した実行履歴を図 3 に示す．

現在，Amida は複数のスレッドによる実行履歴の解析機能を備えていないため，対象がマルチスレッドのプログラムの場合，スレッドごとに実行履歴を記録し，それぞれのシーケンス図を生成，切り替えて表示する．

3.2 圧縮ルール

実行履歴中にはループや再帰構造の中で発生するメソッド呼び出しが全て記録されている．これらをそのままシーケンス図として表現しても，プログラム全体の動作を理解するのは困難である．そこで，実行履歴からこれらを検出，圧縮し，簡潔に図示する．

繰り返し構造と再帰構造を検出，圧縮する方法として，Amida では以下に示す R1 から R4 までのルールを提案している．

- R1: 完全な繰り返し
  実行履歴中の完全に同一な呼び出し構造の繰り返しを圧縮する．
- R2: オブジェクトが異なる繰り返し
  実行履歴中のオブジェクト ID のみが異なる呼び出し構造の繰り返しを圧縮する．
- R3: 欠損構造を含む繰り返し
  実行履歴中の呼び出し構造の一部に欠損を含むような繰り返しを圧縮する．

```
void amida.Main(0).main(java.lang.String[]){
void amida.sequencer.gui.MainFrame(1).<init>(){
void amida.sequencer.gui.SearchDialog(2).<init>(){
amida.sequencer.gui.MainFrame amida.sequencer.gui.MainFrame(0).getInstance(){
}
amida.sequencer.gui.MainFrame amida.sequencer.gui.MainFrame(0).getInstance(){
}
void amida.sequencer.gui.SearchDialog$1(3).<init>(amida.sequencer.gui.SearchDialog){
}
void amida.sequencer.gui.SearchDialog$2(4).<init>(amida.sequencer.gui.SearchDialog){
}
}
void amida.logcompactor.gui.WorkingSetFrame(5).<init>(java.lang.String){
void amida.logcompactor.gui.WorkingSetCanvas(6).<init>(int){
}
void amida.logcompactor.gui.WorkingSetCanvas(7).<init>(int){
}
}
void amida.logcompactor.gui.LogTextAreaFrame(8).<init>(){
void amida.logcompactor.gui.SearchDialog(9).<init>(){
void amida.logcompactor.gui.SearchDialog$1(10).<init>(){
}
void amida.logcompactor.gui.SearchDialog$2(11).<init>(){
}
```

図 3: 実行履歴の例

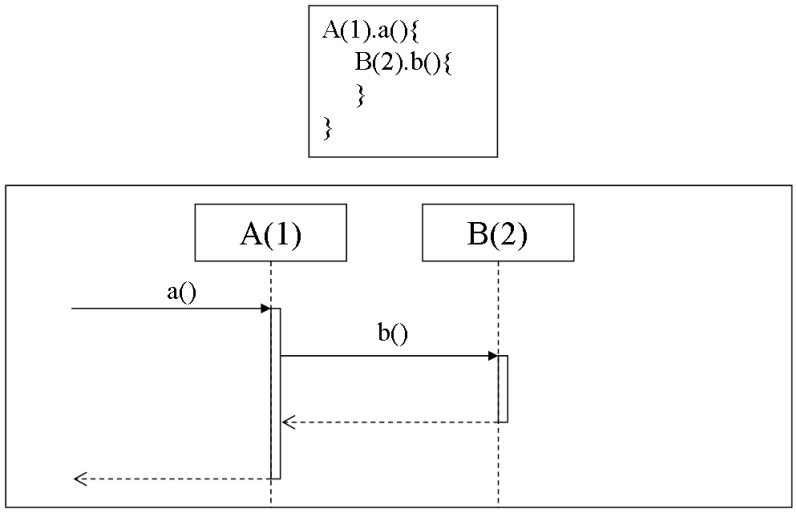

図 4: 未圧縮部から作成される図

- R4: 再帰構造
  実行履歴中の呼び出し構造において再帰的に呼び出されているメソッドの呼び出しを圧縮する．

3.3 シーケンス図の生成

圧縮した実行履歴を基にシーケンス図の生成を行う．また，繰り返し回数などの圧縮結果を基にした情報を注釈として表現し，より分かりやすい図を生成する．

以下，実行履歴中で各圧縮ルールで圧縮された部分について，どのようにしてシーケンス図として生成するかを説明する．

未圧縮部

圧縮ルールを適用しなかった場合や，圧縮ルールを用いても圧縮されなかった部分，圧縮結果を展開した部分については通常のシーケンス図と同様の形式で表現する (図 4).

R1 被圧縮部

R1 によって圧縮された部分には，通常のシーケンスの他にループ情報を表記する (図 5)．なお，このループ情報は以降の R2,R3 についても同様の形式で表現する．

R2 被圧縮部

R2 によって圧縮された部分についても R1 と同様にループ情報の表記を行う．そして，

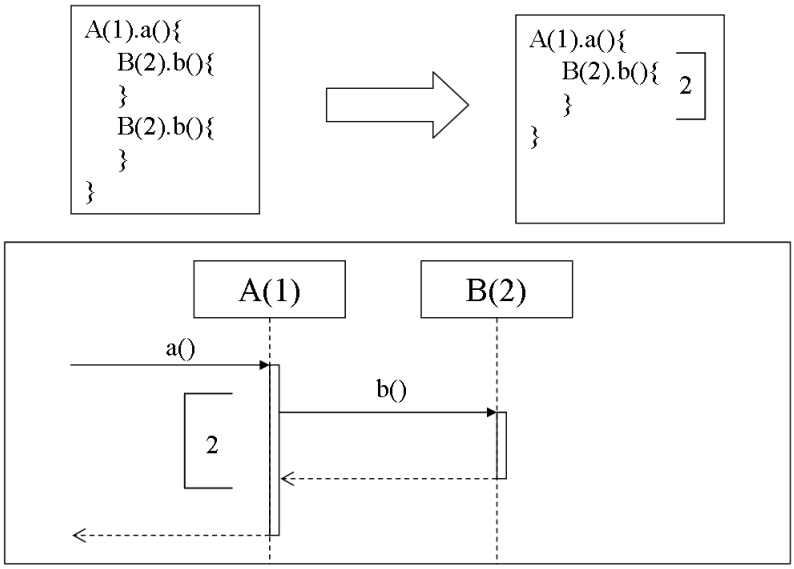

図 5: R1 適用部から作成される図

この部分には複数のオブジェクトを統合したオブジェクトへのシーケンスが存在するため，図中の上部に並ぶオブジェクト列の中に統合されたオブジェクト群を示すオブジェクトを追加し，それに対するシーケンスを引く (図 6).

R3 被圧縮部

R3 によって圧縮された部分にも，ループ情報の表記と統合されたオブジェクトへのシーケンス表現を行う．また，欠損構造と判定されたメソッド呼び出しについては，その呼び出しが行われる場合のシーケンスと，呼ばれずに素通りするシーケンスの 2 通りを記述する (図 7).

R4 被圧縮部

R4 によって圧縮された部分は，統合されたオブジェクトへのシーケンスを含む再帰呼び出しを表す．そのため，再帰呼び出し構造全体を四角で囲み，これを再帰呼び出し全体を表すブロックとする．その内部で発生する，再帰的なメソッド呼び出しは，外側のブロックと同一の名前を持つブロックを内側に作成し，そのブロックへのシーケンスを引くことで，再帰的な呼び出しを表現する (図 8).

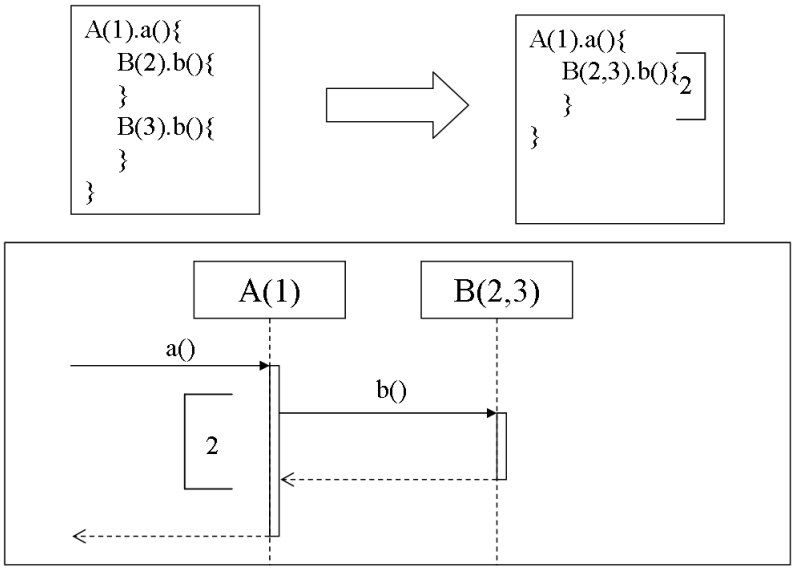

図 6: R2 適用部から作成される図

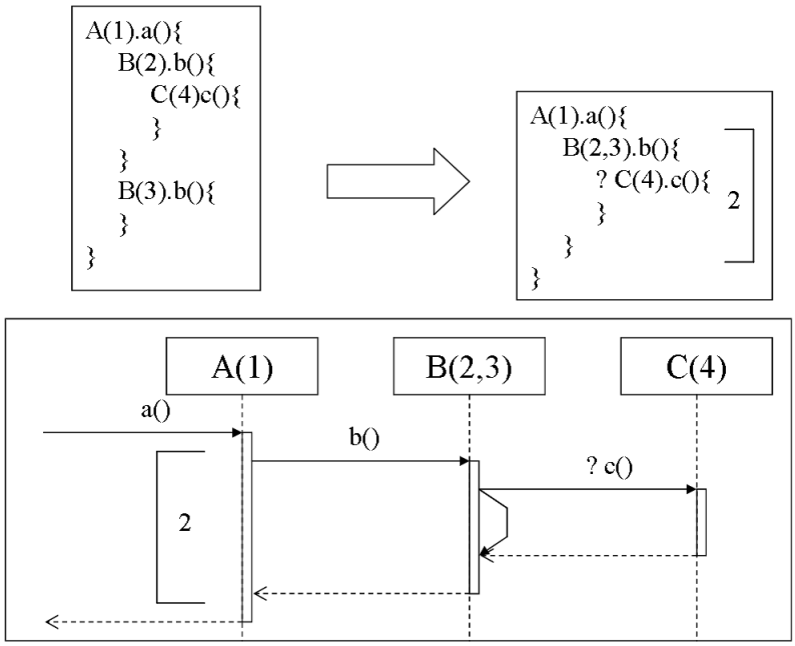

図 7: R3 適用部から作成される図

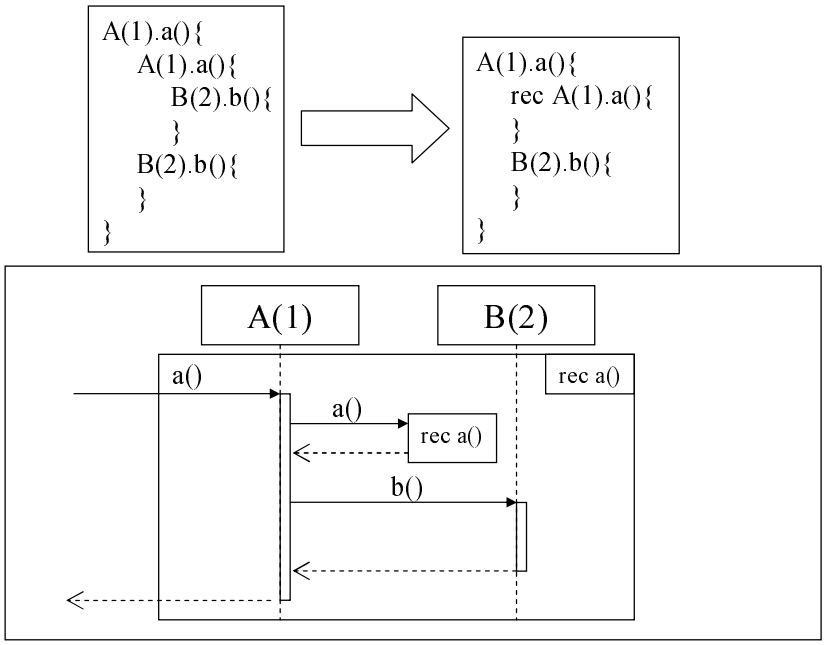

図 8: R4 適用部から作成される図

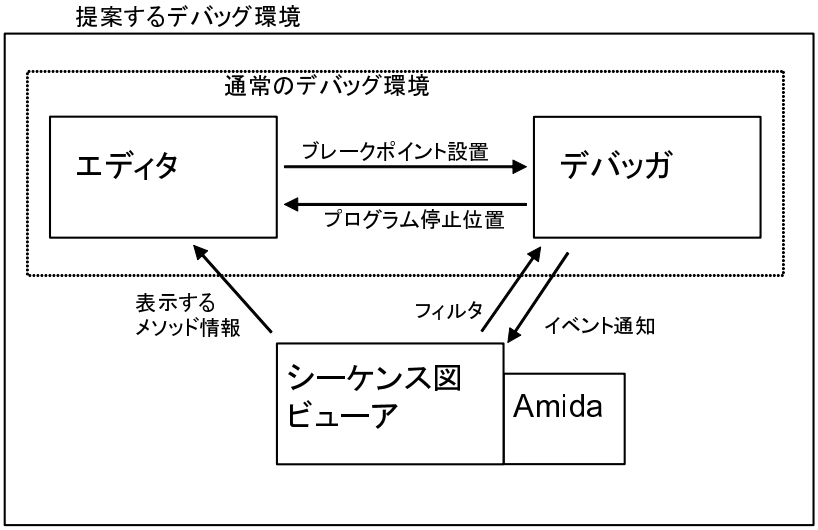

図 9: ツールの概要

4　デバッグ環境への適用

3 章で述べた Amida を使って，実行履歴をシーケンス図として可視化するデバッグ環境を提案 · 試作した．この章では，提案手法とその実装について説明する．

4.1　デバッグ環境

本研究で提案するデバッグ環境は，2.3 節で述べたデバッガの問題を解決するために，デバッグ時に他のデバッグ機能と連動して，実行履歴をシーケンス図として可視化し，各オブジェクトに対するメソッド呼び出しの履歴を示す．これにより，オブジェクトの動作理解を支援する．

具体的には，対象となるプログラムのデバッグ実行時に，メソッド呼び出しの実行履歴を記録し，プログラムのブレークポイントによる停止時，または終了時に，実行履歴からシーケンス図を生成 · 表示する．

ツールの概要を図 9 に示す．このデバッグ環境は，ソースコードを編集するためのエディタ，Java プログラムをデバッグ実行するデバッガ，そしてシーケンス図ビューアで構成される．これらの間の動作を説明する．

デバッグ作業では，エディタ上で利用者がブレークポイントを設置すると，デバッガにそのブレークポイントの情報が渡される．デバッガは，デバッグ実行においてブレークポイントで停止し，そこで停止したことをエディタに渡す．渡されたエディタは，停止したブレークポイントの位置をソースコード上で利用者に示す．ここまでは，通常のデバッグ環境と

同じである．本研究の提案するデバッグ環境ではさらに，デバッガは，デバッグ実行時にメソッドの呼び出しのイベントをシーケンス図ビューアに渡し，それをシーケンス図ビューアでは実行履歴として記録する．また，デバッガはプログラムの停止や終了のイベントをシーケンス図ビューアに渡す．停止，終了のイベントを受け取ったとき，シーケンス図ビューアはその時点までに記録した実行履歴から 3 章で述べた Amida の機能を使い，シーケンス図を作成·表示する．利用者がシーケンス図上でエディタ上に表示したいメソッドを選択すると，シーケンス図ビューアはそのメソッドの情報をエディタに渡し，エディタはそのメソッドを表示する．

本環境を用いて行うデバッグ作業の手順を次に示す．デバッグ作業のうち，既に障害の再現作業は終わっているものとする．

1. ブレークポイントの設定
   デバッグ対象のプログラムの，ソースコード上の任意の位置にブレークポイントを設定する
2. プログラムの実行
   本手法によるデバッグ環境において，プログラムをデバッグ実行する
3. 障害の発生
   デバッグ実行中に取り除きたい障害を発生させる入力や操作を与え，障害を再現する
4. ブレークポイントによる停止，またはプログラム実行の終了
   1 で設定したブレークポイントで停止するか，対象のプログラムが終了する
5. シーケンス図生成
   プログラムの停止または終了に伴い，実行履歴からシーケンス図が生成される
6. バグ位置の推測
   生成されたシーケンス図と設計との比較，または不正な状態のオブジェクトに対するメソッド呼び出し履歴の参照により，障害の原因となり得るメソッド呼び出しを推測する
7. メソッド宣言部のソースコードを確認，修正
   バグ位置と推測されたメソッドについてソースコードを確認，実際にバグがあれば修正する

以上の手順を行うために必要な機能を次に示す．

実行履歴圧縮ルールの切り替え

プログラムのオブジェクト群の動作経路を理解したい場合，シーケンス図はできるだけ簡潔に表示されることが望ましい．

また，あるオブジェクトに対するメソッド呼び出し履歴は，シーケンス図上でそのオブジェクトについて，時間軸の過去の方向へさかのぼっていくことで得られる．しかし，Amida での圧縮ルール R2,3,4(3.2 節) を適用している場合，オブジェクトが統合されることによって，1 つのオブジェクトが図中ではそのオブジェクト単体と，統合されたオブジェクト群の 2ヶ所に出現することがある．このため，あるオブジェクトに対するメソッド呼び出しが，図中では 2 つのオブジェクトへの呼び出しに分散する可能性がある．対象となるオブジェクトがこの状況にあるときは圧縮ルールを R1 のみにする必要がある．このため，適用する圧縮ルールを，任意に選択できる機能が必要である．

図中の各オブジェクトの非表示設定

オブジェクト群の動作経路を理解する場合においても特定のオブジェクトに対するメソッド呼び出し履歴を得る場合においても，必要のない情報は消してシーケンス図を簡潔に表示する方がよい．例えば，シーケンス図上でメッセージが，理解に必要のないオブジェクトをまたがって表示される場合には，その必要のないオブジェクトを表示しない方が理解しやすい．このため，図中の各オブジェクトを，表示しないように設定できる機能を付ける．

この機能を用いたとき，メッセージの送信先のオブジェクトが非表示に設定されている場合は，それ以降のシーケンスが表示されなくなる．また，メッセージの送信元のオブジェクトが非表示に設定されている場合は，どのような経路を経てそのメッセージが送信されたのかが分からない．このため，送信側と受信側のオブジェクトのどちらか片方が非表示に設定されている任意のメッセージについて，そのメッセージに対する操作から，非表示に設定されているオブジェクトを表示状態に戻すことができる機能を付ける．

実行履歴のフィルタ

シーケンス図上には利用者にとって必要のない情報は表示せずに簡潔な図を表示する方がよい．例えば，デバッグを行う開発者にとっては，String クラスや File クラスのような，Java 言語では頻繁に使用されるクラスのオブジェクトについて，動作を理解する必要があるケースは少ない．

このため，実行履歴を取得する段階において，任意のクラスを指定することで，その

クラスのオブジェクトに対するメソッド呼び出しを実行履歴として取得しない機能を付ける．

図中のメッセージからメソッド宣言部へのジャンプ

シーケンス図からバグを含むと推測されたメソッドについて，そのメソッドの宣言部，あるいはそのメソッドの呼び出し元になっているメソッドの宣言部のソースコードを確認し，さらに詳細なバグを特定し修正しなければならない．そのため，指定されたシーケンス図上のメソッド呼び出しのメッセージを操作することで，エディタにそのメソッドの情報を渡し，そのメソッドの宣言部を表示させる機能を付ける．

4.2 実装

以上の手法を基に，ツールの実装を行った．

今回，統合開発環境として Eclipse[2] を選択し，その Java 開発プラグインである Java Development Tools(JDT) プラグインを利用して，Eclipse にプラグインとしてこの手法を組み込むという方法で実装した．実装に用いた言語は Java 言語で，ソースコードは約 6000 行となった．

Eclipse はオープンソースの統合開発環境であり，Java で記述したプラグインを追加することでエディタやコンパイラなどの機能を拡張することができる．いくつかの JDT プラグインが既に用意されており，それらのソースエディタ機能やデバッグ機能を利用して，実行履歴取得機能とシーケンス図表示機能を追加する形式で実装を行った．ツールの構成を図 11 に示す．

プラグインは，デバッグ実行部，シーケンス図表示部，シーケンス図のデータ管理部から構成される．デバッグ実行部によって，デバッグ対象のプログラムを実行，実行履歴を取得する．また，プログラムの停止または終了イベントをデータ管理部に通知する．データ管理部では，デバッグ実行部からのプログラムの停止または終了イベントにより，取得した実行履歴を Amida に渡し，シーケンス図データを生成する．シーケンス図表示部では，データ管理部のシーケンス図データを表示する．また，この表示部を利用者が操作することによって，データ管理部はシーケンス図データの圧縮，オブジェクトの非表示などの処理を行う．

以下では，それぞれの詳細について説明を行う．

デバッグ実行部

Eclipse に用意されている，デバッグ実行を行う JDT プラグインを拡張して，新しいデバッグ実行モードを追加する．追加する実行モードは，標準の Java デバッグモードに実行履歴取得機能と，停止，終了通知機能を付けたものである．これによって，利用者は Eclipse にて他の実行モードを選ぶのと同じように，本ツールのデバッグ実行部を起動することができる．

Java バーチャルマシンからイベントを受け取るには，Java Debug Interface(JDI)[3] を使う．この機能を使ってバーチャルマシンから，メソッドの開始や終了，例外発生，ブレークポイントでの停止，スレッドの終了などの各イベントの通知を受ける．

この JDI の機能を使って，デバッグ対象のプログラムのメソッド開始時と終了時に実行履歴を出力する．出力した実行履歴は，データ管理部を通して Amida に渡される．実行履歴は 3.1 節の Amida の形式に従い，メソッド開始時には「**返り値の型 クラス名 (オブジェクト ID). メソッド名 (引数の型)**{」，メソッド終了時には「}」を記録する．また，Amida と異なり，例外が発生してメソッドが終了する場合には「}**(例外名)**」を記

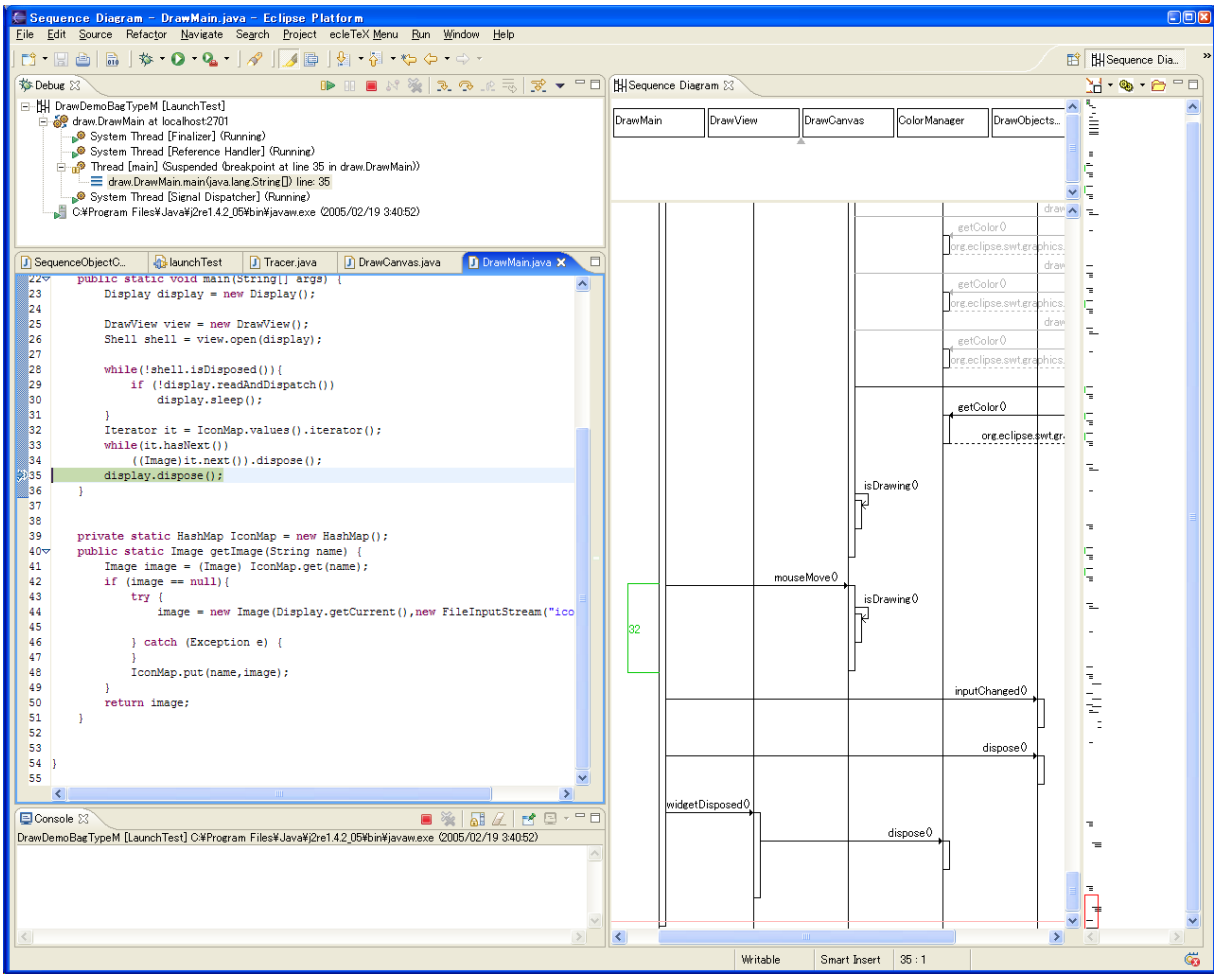
Sequence Diagram – DrawMain.java – Eclipse Platform
File Edit Source Refactor Navigate Search Project ecleTeX Menu Run Window Help
Debug
DrawDemoBagTypeM [LaunchTest]
draw.DrawMain at localhost:2701
System Thread [Finalizer] (Running)
System Thread [Reference Handler] (Running)
Thread [main] (Suspended (breakpoint at line 35 in draw.DrawMain))
draw.DrawMain.main(java.lang.String[]) line: 35
System Thread [Signal Dispatcher] (Running)
C:¥Program Files¥Java¥j2re1.4.2_05¥bin¥javaw.exe (2005/02/19 3:40:52)
SequenceObjectC...
launchTest
Tracer.java
DrawCanvas.java
DrawMain.java
Console
DrawDemoBagTypeM [LaunchTest] C:¥Program Files¥Java¥j2re1.4.2_05¥bin¥javaw.exe (2005/02/19 3:40:52)
Sequence Diagram
DrawMain
DrawView
DrawCanvas
ColorManager
DrawObjects...
getColor()
isDrawing()
mouseMove()
inputChanged()
dispose()
widgetDisposed()
Writable
Smart Insert
35 : 1

図 10: 実行画面

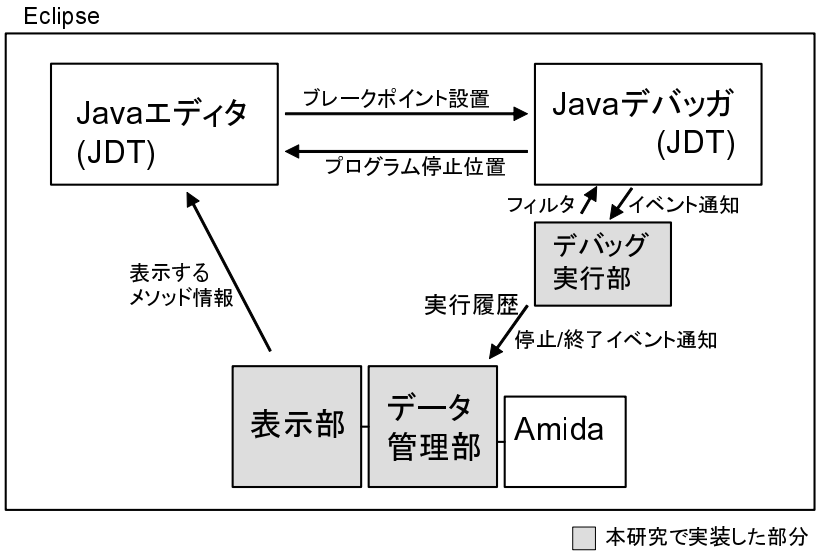

図 11: ツールの構成

録する．対象がマルチスレッドのプログラムの場合にはスレッド別に実行履歴を記録する．JDI に，”*”をワイルドカードとして用いてクラス名を指定することで，そのクラス名に一致するクラスのオブジェクトについて，メソッド呼び出しを取得しないようにすることが可能である．これを利用して実行履歴のフィルタ機能を実現している．

また，JDI の機能を使って，デバッグ対象プログラムのブレークポイントでの停止，または終了イベントを受け取り，シーケンス図データ管理部へ通知する．

他の実行モードと同様，実行前に実行の設定を行うダイアログにて各種設定を行うことができるが，このダイアログに実行履歴のフィルタ機能の設定を行うタブを追加し，利用者が任意に実行履歴を記録しないクラスを選択，追加できるようにしている．

シーケンス図表示部

Eclipse のユーザインタフェイスの機能を拡張して，シーケンス図を表示するビューを追加する．このビューは，データ管理部から渡されたシーケンス図データをシーケンス図として表示するものである．前述のデバッグ実行と連動してこのビューが統合開発環境に表示される．

図 12 にシーケンス図表示ビューのスクリーンショットを示す．シーケンス図表示ビューはシーケンス図の表示を行うオブジェクト表示部，シーケンス表示部，縮小表示部の 3 つと，その他各機能を操作するボタンにより構成される．縮小表示部とはシーケンス図全体を縮小表示する部分で，オブジェクト表示部とシーケンス表示部が表示して

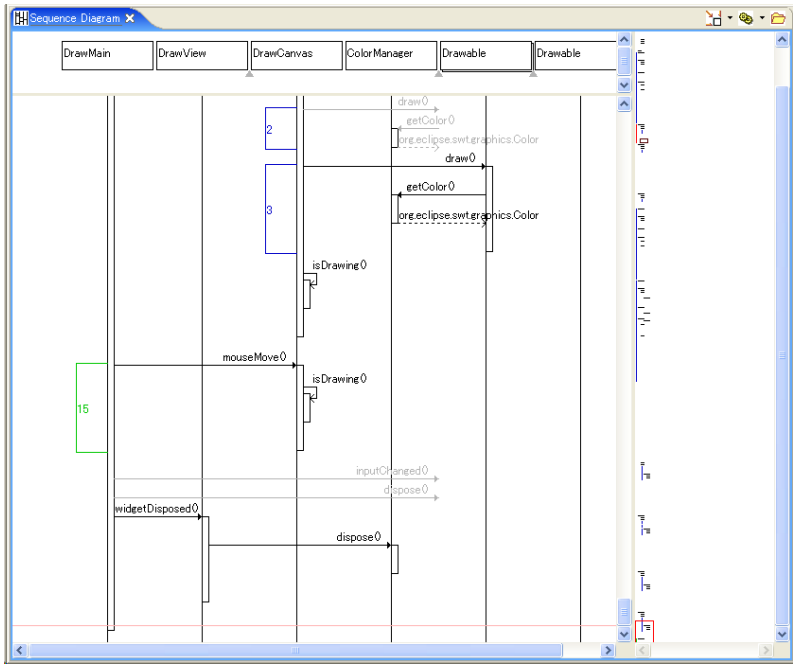

図 12: シーケンス図表示ビュー

いる範囲を赤枠で示すことで，この 2 つがシーケンス図全体の中でどの辺りを表示しているのかが分かるようにしている．

4.1 節で示した機能のうち，シーケンス図表示部から操作できる機能は，実行履歴圧縮ルールの切り替え，図中の各オブジェクトの非表示設定，図中のメッセージからメソッド宣言部へのジャンプである．

実行履歴圧縮ルールの切り替え

シーケンス図表示ビューの上部に位置するメニューにより，実行履歴に適用する圧縮ルールを任意に選択できる．利用者が適用する圧縮ルールを変更した場合，この設定情報をデータ管理部に渡し，それまで表示していたシーケンス図のデータに，設定した圧縮ルールを適用，結果を再度表示する．

図中の各オブジェクトの非表示設定

図中の任意のオブジェクトを非表示にするように指定できる．利用者が設定を変更した場合，データ管理部がシーケンス図データを再構成，結果を再度表示する．

非表示に設定されたオブジェクトが存在することを示すために，本来存在する位置の左右に表示されているオブジェクトの間にマーカ (▲印) を表示する．非表示に設定されているオブジェクト間のメッセージは，シーケンス図上では表示しない．また，送

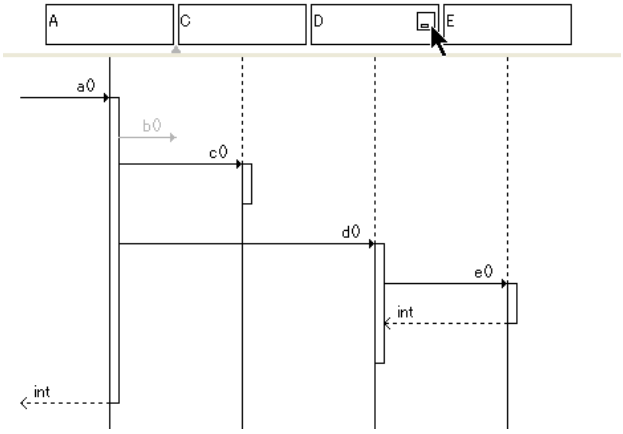

図 13: オブジェクトの非表示設定 (設定前)

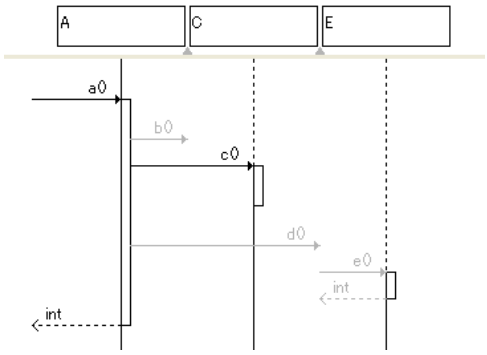

図 14: オブジェクトの非表示設定 (設定後)

信元または送信先のオブジェクトのどちらか片方が非表示であるようなメッセージは，シーケンス図上では薄く表示する．これを半表示状態と呼ぶこととする．半表示状態のメッセージは非表示に設定されているオブジェクトが本来存在すべき位置と，もう片方の表示されているオブジェクトの間に矢印を引いて表示する．

表示中のあるオブジェクトを非表示に設定する場合，そのオブジェクトを表す四角の枠内にマウスカーソルを移動すると，図 13 のように非表示ボタンが現れるので，そのボタンを押す．するとそのオブジェクトが非表示に設定される．

1 つのオブジェクトを除く全てのオブジェクトを非表示に設定する場合，そのオブジェクトの枠内をダブルクリックする．するとそのオブジェクトを除く全てのオブジェクトが非表示に設定され，図 14 のような表示となる．

非表示に設定されたオブジェクトを表示する設定に戻す場合，非表示に設定されてい

るオブジェクトの存在を示すマーカを操作する方法と，半表示状態のメッセージを操作する方法がある．マーカを操作する場合は，マーカを右クリックするとメニューが現れ，その位置に隠れているオブジェクトの一覧が表示されるので，その中から表示したいオブジェクトを選ぶ方法と，マーカをダブルクリックして，その位置に隠れているオブジェクトすべてを表示する方法がある．半表示状態のメッセージからたどる場合は，半表示状態のメッセージ上でダブルクリックすれば，そのメッセージの送信先または送信元のオブジェクトのうち，非表示に設定されているオブジェクトが表示される．

また，ブレークポイントで停止中のシーケンス図では，停止位置からさかのぼって数回分のメソッド呼び出しメッセージの，メソッド呼び出し元または呼び出し先となっているオブジェクトのみ，初期状態で表示し，他は非表示に設定する．

図中のメッセージからメソッド宣言部へのジャンプ

シーケンス図上のメソッド呼び出しメッセージからソースコード上のメソッド宣言部へジャンプできる．図上に表示されているメソッド呼び出しメッセージをダブルクリックすると，対応するメソッドの宣言部がソースエディタ上で開かれる．これは，JDTの検索機能を使って，クラス名，メソッド名，引数の型からメソッド情報を検索，検索結果のメソッド情報をソースエディタに渡すことで実装した．

また，他に次のような機能を実装している．

Amida には複数のスレッドの実行履歴解析機能はないため，対象がマルチスレッドのプログラムの場合は複数のシーケンス図を生成する．そのため，複数のスレッドのシーケンス図から，表示する図を選択することができる．

また，シーケンス図表示ビューでは，オブジェクトのクラス名やメッセージのメソッド名と引数など，多くの文字列を表示すると図が見づらくなる．このため，図には最低限の文字列のみ表示して詳しい情報はツールチップを使って表示する．ツールチップとは，利用者が画面上のある領域にマウスカーソルを重ねると表示される，ヒントを表示するウィンドウのことである．

シーケンス図のデータ管理部

データ管理部では，Amida を使ったシーケンス図データの生成と，生成したデータへの各種操作を行う．

まず，デバッグ実行部によりブレークポイントによる停止または終了が通知されると，Amida を用いて実行履歴を解析，圧縮し，シーケンス図データを生成する．そして，

このデータに対してオブジェクトとメッセージの座標，オブジェクトとメッセージの非表示設定，表示する文字列などの情報を付加し，シーケンス図表示部に渡す.

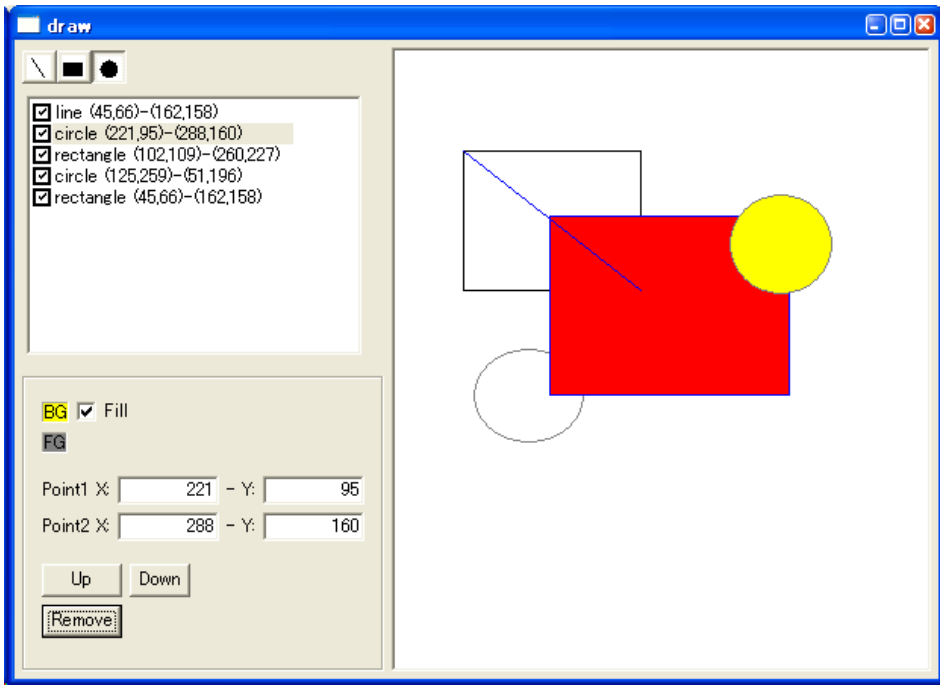

図 15: ドローツール

5　適用例

4 章にて述べた試作環境を用いて，実際に Java プログラムの開発を行い，シーケンス図のデバッグにおける有効性を確認した．

5.1　実験

5.1.1　開発したプログラム

実験の対象として，図形の作成，描画を行う簡単なドローツールを開発した．開発に用いた言語は Java 言語で，GUI のライブラリとして SWT を用いた．ドローツールのスクリーンショットを図 15 に示す．以下，このドローツールについて簡単に説明する．

図形の種類

本ツールが描画する図形は，2 頂点の間の直線，またはその直線を対角線とし，X 軸 Y 軸に平行な辺を持つ長方形，またはその長方形に内接する楕円の 3 種類である．図 15 に示す画面左上のボタンによって，配置する図形の種類を選択する．

図形の配置と描画

図形の配置は画面右側の図形描画部で行う．図形描画部上でマウスのボタンを押し，

ボタンを押したままカーソルを移動し，ボタンを離すと，マウスのボタンを押した時のカーソルの座標，頂点 1 から，ボタンを離したときのカーソルの座標，頂点 2 までの 2 点間で，図形が配置，描画される．

また，図形描画部に配置された図形は，画面左上の図形リストに表示される．リスト中の上下は図形の重なりに対応している．つまり，リストの上部にあるものほど前面に表示される．また，リストには，それぞれの図形を表すテキストの左側にその図形を表示するかしないかを指定するチェックボックスがあり，このチェックボックスにチェックの入っている図形のみ，描画部に表示される．

図形パラメータの設定

パラメータ設定エリアでは，現在設定対象となっている図形の，2 頂点の X 座標，Y 座標それぞれに対応する入力ボックスに入力することで図形の描画位置を変更することができる．また，FG ボタン，BG ボタンを押すことで表示される色指定ダイアログを用いて，枠線の色や塗りつぶす色を指定することができる．設定対象の図形とは，図形リスト中で選択された図形，または図形描画部にマウスによって新たに配置された図形である．新たに別の図形が設定対象となった時，パラメータ設定エリアの色指定ダイアログを表示するボタンと頂点座標の各入力ボックスには，その新たに設定対象になった図形の値が入力される．プログラムの開始直後で設定対象がない場合は，各入力ボックスに初期値として 0 が入る．

また，Up ボタン，Down ボタンによる重なりの順序の変更や，Remove ボタンによる図形の削除も行うことができる．

5.1.2 デバッグ作業

開発途中において，マウスによって図形を配置する際に，頂点の座標が別の値に変更されてしまう，という不具合が発生した．図形を配置すると，マウスのボタンを離した直後に，図形がリストに追加され，パラメータ設定部の設定対象となるものの，頂点座標の X と Y の値が，頂点 1 の X 座標以外すべて 0 に変わってしまった．

この不具合を修正するために，試作したデバッグ環境を利用してデバッグ作業を行った．まず，図形配置処理の動作の設計を表すシーケンス図と，この不具合が発生した際に取得した実行履歴から生成されたシーケンス図との比較を行った．図 16 は設計時の図であり，図 17 は，生成された図から動作の理解に必要な箇所だけを抜き出し，メソッド名やクラス名を日本語に置き換えたものである．図形配置処理の動作の設計では，マウスのボタンを離すと，2 つ目の頂点の座標の指定メソッドが呼ばれ，その後，リストへ追加とパラメータ設定部の更新を行う，というものである．生成されたシーケンス図では，マウスのボタンを離し

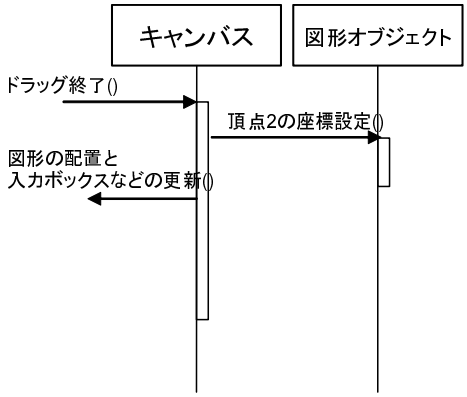

図 16: 図形配置処理の動作設計

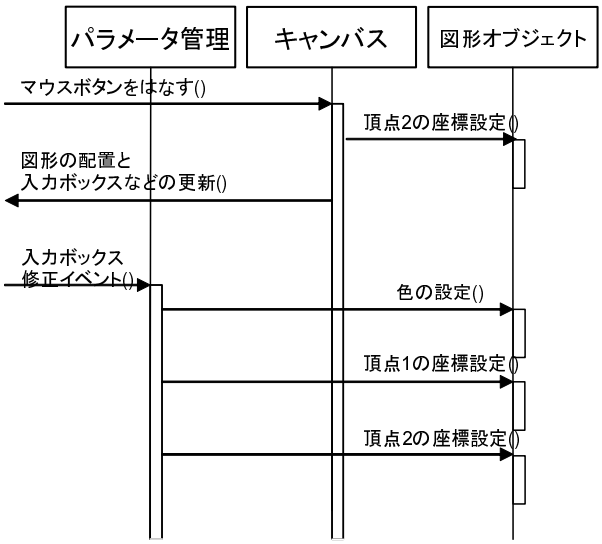

図 17: 図形配置処理の実際の動作

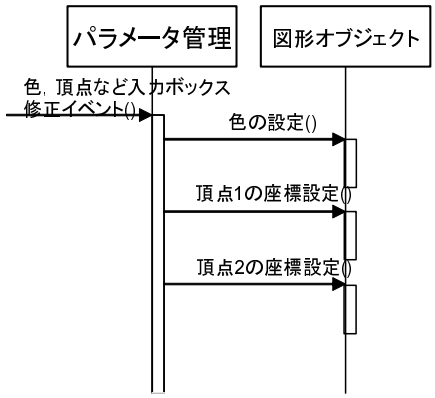

図 18: 図形パラメータ設定処理の動作設計

た後，2 つ目の頂点の座標の指定メソッドが呼ばれる．次に，リストへ追加とパラメータ設定部の更新を行う．ここまでは設計通りである．しかし，設計には存在しない，入力ボックスの修正イベントが呼び出され，その中で頂点の指定メソッドが呼び出されていることが分かった．図 18 に，入力ボックスの修正での図形のパラメータ設定処理の動作設計を示す．これは，利用者による修正を想定して設計したものである．このパラメータ設定処理が，マウスによって図形を配置した直後のパラメータ設定部の更新により引き起こされていた．

これにより，1 つ目の入力ボックスの値を新たに配置した図形の値へ変更した時，入力ボックスの修正イベントが発生し，まだ新しい図形の値に変更していない他の入力ボックスの値が図形の頂点の座標に反映されていた．

このような動作が発生しないようにするために，マウスによる新たな図形の配置と，利用者による入力ボックスからの図形の描画位置の修正との区別を行い，新たな図形を配置したことによる入力ボックスの値の更新では，図形の描画位置の修正イベントを発生させないようにすることで，不具合を修正した．

5.2 評価と考察

開発者は本デバッグ環境において，実行履歴から作成されたシーケンス図から実際の動作を理解し，バグの位置を推測することができた．

この実験から，設計と実際の動作との比較は，デバッグに有効であることが確認できた．

ただし，シーケンス図には各オブジェクトのクラス名と呼び出されるメソッド名だけが表示されることから，対象のプログラムのクラスやメソッドについて，それらがどのような処理を行うか知っている必要がある．試作環境では，図中のメソッド呼び出しのメッセージから，ソースコードの該当メソッドの宣言部へジャンプする機能を用いることで，そのメソッ

ド内の処理の内容を把握できるようにしている．

また，実行履歴取得部の実行速度が遅いという問題がある．本ツールのデバッグ実行部では，Eclipse に標準で用意されているデバッグ実行と比べて 10 倍以上の時間がかかった．この点については，今後改善していく必要がある．

6　まとめ

デバッグにおいて，バグ位置を特定する作業を支援するために，プログラム実行履歴からシーケンス図を生成する機能を統合開発環境 Eclipse のデバッガに組み込んだ．試作したデバッグ環境により，デバッグ実行機能と連動して表示されるシーケンス図を用いて，対象のプログラムの実際の動作と設計の比較を行うこと，また，任意のオブジェクトに対するメソッド呼び出し履歴を知ることで，バグ位置の推測ができる．また，シーケンス図上で登場したメソッド名からソースコード上のメソッド定義へ移動する機能によって，障害の原因と推測したメソッドについてのソースコードを素早く確認·修正することを可能とした．

試作したデバッグ環境を用いてプログラムの開発を行い，シーケンス図の使用がデバッグに有効であることを確認した．

大規模なプログラムのデバッグにおいては，実行履歴の長さ，登場するオブジェクトの数がともに大きくなり，シーケンス図の可読性が低下する恐れがあるが，障害を再現しないようなテストケースでの実行履歴との差分を利用して手がかりを提示する手法 [11] や，デバッグ操作の列を小さなプログラムとして実行することで半自動的に情報を探索する手法 [5] と組み合わせることによって，大規模ソフトウェアの開発に着目した場合にも，効果的なデバッグが実現できると考えられる．

謝辞

本研究の全過程を通して，常に適切な御指導，御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 井上克郎教授に心より深く感謝致します．

本論文を作成するにあたり，常に適切な御指導，御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 楠本真二助教授に心から感謝致します．

本論文を作成するにあたり，適切な御指導，御助言を賜りました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 松下誠助手に心から感謝致します．

本研究を通して，逐次適切な御指導，御助言を頂きました独立行政法人科学技術振興機構さきがけ研究員 神谷年洋氏に深く感謝致します．

本研究を通して，逐次適切な御指導，御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 石尾隆氏に深く感謝致します．

本研究を通して，逐次適切な御指導，御助言を頂きました大阪大学大学院情報科学研究科コンピュータサイエンス専攻 谷口考治氏に深く感謝致します．

本研究に対して，開発現場の視点から貴重なコメントを頂いた，株式会社日立システムアンドサービス 津田道夫氏，高橋まゆみ氏，英繁雄氏，前田憲一氏に深く感謝致します．

最後に，その他様々な御指導，御助言等を頂いた大阪大学大学院情報科学研究科コンピュータサイエンス専攻井上研究室の皆様に深く感謝いたします．

参考文献

[1] Tankut Akgul, Vincent J. Mooney III, and Santosh Pande: A Fast Assembly Level Reverse Execution Method via Dynamic Slicing. Proceedings of 26th International Conference on Software Engineering(ICSE 2004), pp.522-531, May 2004.

[2] Eclipse.org. *http://www.eclipse.org/*

[3] Java Platform Debugger Architecture.
http://java.sun.com/j2se/1.4.2/docs/guide/jpda/

[4] Bil Lewis and Mireille Ducassé: Using Events to Debug Java Programs Backwards in Time. Companion of 18th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), pp.96-97, October 2003.

[5] Guillaume Marceau, Gregory H. Cooper, Shriram Krishnamurthi, Steven P. Reiss: A Dataflow Language for Scriptable Debugging. Proceedings of 19th IEEE International Conference on Automated Software Engineering(ASE 2004), pp.218-227, September 2004.

[6] Wim De Pauw, David Lorenz, John Vlissides and MarkWegman: Execution Patterns in Object-Oriented Visualization. Proceedings of the 4th Conference on Object-oriented Technologies and Systems, pp.219-234, April 1998.

[7] Steven P. Reiss and Manos Renieris: Encoding program executions. Proceedings of the 23rd International Conference on Software Engineering, pp.221-230, May 2001.

[8] Tamar Richner and Stephane Ducasse: Using Dynamic Information for the Iterative Recovery of Collaborations and Roles. Proceedings of the 18th International Conference on Software Maintenance, pp.34-43, October 2002.

[9] 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 (FOSE2004), pp.5-15, 11 月 2004.

[10] Unified Modeling Language (UML) 1.5 specification. OMG, March 2003.

[11] W. Eric Wong and Yu Qi: An Execution Slice and Inter-Block Data Dependency-based Approach for Fault Localization. Proceedings of 11th Asia-Pacific Software Engineering Conference(APSEC 2004), pp.366-373, November 2004.