修士学位論文

題目

プログラム実行履歴を用いたオブジェクト生成関係の可視化手法

指導教員
井上 克郎 教授

報告者
中野 佑紀

平成 23 年 2 月 7 日

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 ソフトウェア工学講座

平成 22 年度 修士学位論文

プログラム実行履歴を用いたオブジェクト生成関係の可視化手法

中野 佑紀

内容梗概

オブジェクト指向プログラムが，1 つの処理を実行する際には，その中間データを表現する一時オブジェクトを多数生成する．これらのオブジェクトがどのように生成されるかを知ることは，実行している処理の内容を理解する上で有用である．しかし，オブジェクト指向プログラムにおいて，オブジェクトがどのように生成されるかをソースコードのみから調査することは容易でないため，どのようにオブジェクトが生成されるかを効果的に調査する方法が必要である．

そこで本研究では，オブジェクト指向プログラムを実行することで取得した実行履歴を解析することにより，実際のプログラム実行におけるオブジェクト生成関係を可視化する手法を提案する．具体的には，オブジェクトの生成処理において，あるオブジェクトがどのオブジェクトにより生成され，また，どのオブジェクトを利用して生成されたかをオブジェクト生成関係として特定し，有向グラフとして可視化する．Java プログラムを解析対象として，提案手法をツール化し，プログラム理解における利用方法を示した．また，ライフゲームプログラムを対象として，メソッドのテストに必要なオブジェクトの生成を行い，61 メソッドのうち，46 メソッドのテストに必要なオブジェクトを生成することができた．オブジェクト生成関係の特定にかかる時間とグラフの集約の効果を調査した．その結果，利用を想定しているオブジェクト数ではオブジェクト生成関係を現実的な時間で特定することができ，集約は本手法で提示するグラフのサイズを抑えるために有効であることが分かった．

主な用語

プログラム理解
オブジェクト指向プログラム
動的解析
データフロー解析
オブジェクト生成関係
可視化

目 次

1　まえがき

ソフトウェアのデバッグや保守作業ではプログラム理解が重要である．開発者は作業における多くの時間をプログラム理解に費やしている [11]．そのため，開発者の負担を軽減するには，プログラム理解の支援が必要である．

プログラムは，既存のデータ群を用いて新たなデータを生成することの繰り返しで処理を実現している．オブジェクト指向プログラムにおいては各データはオブジェクトによって管理されており，プログラムが 1 つの処理を実行する際には，その中間データを表現する一時オブジェクトが大量に生成されている [13]．これらのオブジェクトに格納されたデータを知ることは，実行している処理内容を理解する上で有用である．実際に LaToza らによる開発者へのアンケート調査では [11]，開発作業において，オブジェクトが生成された手順を調べることが多いと指摘されている．

オブジェクト指向プログラムにおいて，プログラム実行のある時点におけるオブジェクトがどのように生成されるかをソースコードのみから調べることは容易でない．これは，オブジェクト生成とは，プログラミング言語 Java における new 演算子のようにオブジェクト用のメモリを確保して，コンストラクタによる初期化を実行するだけでなく，そのあとにオブジェクトに適切なデータを格納するための一連の処理が続く場合があるためである．たとえば，予定管理プログラムにおいて，ファイルに保存された予定データを読み込み,「予定のリスト」を表現したオブジェクトを生成するとする．この読み込み処理では，まず空のリストを生成し，そのリストに対して，ファイルの内容から生成された予定データオブジェクトを追加していくことになる．開発者がこの処理を理解しようとすると，ファイルのデータがどのようにこの処理に渡され，ファイルからどのような予定データが生成され，そしていつ予定のリストが完成したかを知る必要がある．オブジェクト指向プログラムでは，動的束縛を使用すると，プログラムの実行の流れが実行時まで確定せず，また，それを活用した動的なオブジェクトの生成方法も Factory パターンとして知られているため [8]，単純にソースコードを追跡するだけでは，生成手順の理解が困難な場合もある．オブジェクトがどのように生成されたかを正確に知るには実行時の動作を観測することが効果的である．たとえば，デバッガとブレイクポイントを用いて実行時の動作を観測することで，どのようにオブジェクトが生成されるかを調べることができる．開発者は調査対象のオブジェクトの生成処理が開始される前の時点でプログラム実行を一時停止し，ステップ実行を繰り返すことで，プログラムの 1 ステップの実行ごとにオブジェクトがどのように生成されるかを調査する．しかし，オブジェクトの生成に，他のオブジェクトが利用されている場合，利用されているオブジェクトについても同様にさかのぼって生成処理を調査していく必要があるため，調査すべきオブジェクトの生成処理の数は膨大になる可能性がある．また，巨大なプログラムを 1 ス

テップずつステップ実行していくことは現実的でないため，どのようにオブジェクトが生成されるかを効果的に調査する方法が必要である．

そこで本研究では，オブジェクト指向プログラムの実行時情報を解析することにより，実際のプログラム実行でオブジェクトがどのように生成されたかを可視化する手法を提案する．具体的には，プログラム実行時に行われた内容を実行順に記録した実行履歴を解析することにより，オブジェクトの生成処理において，あるオブジェクトがどのオブジェクトにより生成され，また，どのオブジェクトを利用して生成されたかをオブジェクト生成関係として特定する．そして，オブジェクトを頂点，オブジェクト生成関係を辺とする有向グラフとして可視化する．これにより，オブジェクト生成においてどのオブジェクトがどのオブジェクトに影響しているかを容易に理解できる．本手法を用いることにより，オブジェクトがどのように生成されるかを調べる作業を支援できると考えている．

Java プログラムを解析対象として，提案手法をツール化し，プログラム理解における利用方法を示した．また，ライフゲームプログラムを対象として，メソッドのテストに必要なオブジェクトの生成を行い，61 メソッドのうち，46 メソッドのテストに必要なオブジェクトを生成することができた．オブジェクト生成関係の特定にかかる時間とグラフの集約の効果を調査した．その結果，利用を想定しているオブジェクト数ではオブジェクト生成関係を現実的な時間で特定することができ，集約は本手法で提示するグラフのサイズを抑えるために有効であることが分かった．

本論文の構成は次の通りである．まず，2 章で本研究の背景について述べ，3 章で提案手法について説明する．4 章では提案手法の実装について説明し，5 章では実施したケーススタディについて説明する．6 章では関連研究を紹介する．最後に 7 章で本研究のまとめと今後の課題について述べる．

2　背景

ソフトウェアのデバッグや保守作業ではプログラム理解が重要である．プログラム理解とは，対象となるプログラムの動作や構成，プログラムを構成しているコンポーネントの使用方法などを理解することである．多くの場合，開発者はプログラムを理解するためにドキュメントの読解を試みる．しかし，ドキュメント中に理解しようとしている内容が記述されているとは限らず [7, 23]，記述されていたとしてもドキュメントの更新を怠っていると実際のプログラムとの差異が生じる [4, 18]．そのため，プログラムを解析することでプログラム理解を行うことが必要である．

プログラム解析は，静的解析と動的解析に大別できる．静的解析は主にソースコードを対象として解析を行うため，プログラム全体を解析することができるが，静的構造と実行時動作の間にギャップが存在するとプログラムを理解することが困難になる．このギャップの例として，オブジェクト指向における動的束縛があり，静的解析によりプログラム理解を正確に行うことは容易ではない [12, 14]．動的解析はプログラム実行時に取得できる情報を対象として解析を行うため，実際に実行された内容しか理解することができず，プログラムの理解したい部分を実行するための実行シナリオが必要になるが，実行時の動作を詳細に理解することができるため，オブジェクト指向プログラムのプログラム理解には動的解析を用いることが有効である．

動的解析を利用したプログラム理解支援手法として，実行時に発生したメソッド呼び出し列を UML のシーケンス図として可視化する手法 [25] やオブジェクトがどのオブジェクトを所持しているかを可視化する手法 [20]，特定のオブジェクトへの操作を可視化する手法 [19]，オブジェクトの動作モデルを作成する手法 [7] などが提案されている．

2.1　データフロー解析

本研究は，動的解析を用いたデータフロー解析の一種である．データフロー解析は，プログラム中でデータがどのように伝播しているかを解析する．データフロー解析を用いることで，生成されたデータがどこで利用されるか，どのようにデータが移動したか，どのデータがどのデータに影響したかなどを調べることができる．データフロー解析は，動的解析よりも静的解析の分野で広く用いられている [14, 15]．

オブジェクト指向プログラムに対するデータフロー解析の一種として，オブジェクトフロー解析が提案されている [14]．オブジェクトフロー解析では動的解析を用いることにより，プログラム実行中にどのオブジェクトがどのような経路で渡され，どのオブジェクトで保持されたかを解析している．このオブジェクトフロー解析を用いるものとして，どの機能で生成されたオブジェクトをどの機能で利用しているかを特定することで機能間の関係を特定す

る手法 [16] や，ある特定のメソッド内で必要な参照関係とそのメソッドの実行により変化する参照関係を明らかにする手法 [15] が提案されている．

他にもデータフロー解析を利用する手法として，プログラム中の余分なデータコピーを検出するためにオブジェクト間のデータコピーの流れを調べる手法 [24] やデータの不適切な利用を検出するための手法 [5] などが提案されている．

2.2 オブジェクト生成

オブジェクトは，フィールドを用いてデータを管理している．フィールドで管理されているデータは，オブジェクトに対する操作によって内容が変更される．そのため，同一のオブジェクトであっても，プログラムの実行時点によってフィールドで管理されているデータの内容が異なる場合がある．フィールドがオブジェクトの状態を表していると考えると，すべてのフィールドのデータが一致するオブジェクトを同じ状態を持つオブジェクトとみなすことができる．そこで，本研究ではオブジェクト生成とはオブジェクトを構築し，適切なデータを格納することで，ある特定状態のオブジェクトを生成することとする．

オブジェクト指向においてクラスは基本的なコンポーネントであり，クラスの使用方法を理解することは重要である [22]．クラスの使用方法を知る際はそのクラスで実装されているメソッドを理解する必要がある．メソッドを実行するには，呼び出し対象となるレシーバ (receiver object) や引数として用いる適切な状態のオブジェクトを用意する必要がある．そのため，メソッドを理解する際にはレシーバや引数のオブジェクトが満たすべき条件を知る必要がある．しかし，実際にそのメソッドを利用する際にはレシーバや引数のオブジェクトが満たすべき条件を知っていたとしても，そのオブジェクトの生成方法を知らなければ，メソッドを実行することはできない．

ここで，オブジェクトの生成方法を提示する既存手法とその問題点について説明する．オブジェクトの生成方法を知る手法として，利用方法の分かっているクラスのオブジェクトから利用したいクラスのオブジェクトを取得するメソッド列を生成する手法が提案されている．具体的には，API を用いてメソッド列を生成する手法 [17] やコードサーチエンジンを用いて取得したコードサンプルからメソッド列を生成する手法 [23] が提案されている．しかし，これらの手法は静的に行われているため，メソッド列を生成できない可能性があり，生成できたとしても利用したいクラスのオブジェクトが必ず取得できるとは限ず，利用したいクラスのオブジェクトを取得できたとしてもオブジェクトの状態が利用目的に合わない可能性もある．そこで，本研究ではプログラム実行中に利用方法を知りたいメソッドの実行時に利用されているオブジェクトがどのように生成されるかを知ることでクラスの使用方法の理解に役立つと考えた．

また，プログラムが 1 つの処理を行う際には，その中間データを表現する一時オブジェク

トが大量に生成されているため [13]，その役割を知ることでその処理の内容を理解できると考えられる．しかし，同じクラスのオブジェクトでも状況によってふるまいが変化するため [21]，どのクラスのオブジェクトが生成されるかを知るだけではオブジェクトの役割を判断することは容易でない．そこで，処理の実行中に生成されるオブジェクトがどのように生成され，どのオブジェクトの生成に影響しているかを知ることで，オブジェクトの役割を知る手掛かりになり，処理の内容を理解することができると考えた．

ある状態のオブジェクトを生成するには，オブジェクト用のメモリ領域を確保した上で，複数の操作を行う必要がある．また，操作を行う際に利用するオブジェクトに対しても，その時点における状態のオブジェクトの生成方法を知ることが必要になる．そのため，オブジェクトの生成処理が膨大になることが考えられ，その処理を逐一確認していくことは現実的でない．

そこで．本研究ではプログラムの実行時情報を解析し，指定した時刻における状態のオブジェクトがどのようにして生成されたかを可視化する．具体的にはプログラム実行時に行われた内容を実行順に記録した実行履歴を解析し，どのオブジェクトにより構築され，また，オブジェクト生成処理においてどのオブジェクトを利用したかをオブジェクト生成関係として特定する．そして，オブジェクトを頂点，オブジェクト生成関係を辺とする有向グラフとして可視化する．これにより，指定した時点における状態のオブジェクトの生成において，どのオブジェクトが影響しているかを容易に理解できる．

3　提案手法

本研究では，対象とするプログラムを実行することで取得した実行履歴を解析し，開発者の指定した実行履歴上の区間におけるオブジェクト生成関係を特定し，有向グラフとして可視化する手法を提案する．また，指定区間内における特定のオブジェクトの生成に注目するため，特定した生成関係から指定したオブジェクトの生成に関わる部分グラフを抽出することが可能である．なお，本手法は Java プログラムを対象としている．

この章では解析に用いる実行履歴と実行履歴に記録する情報について説明し，その後オブジェクト生成関係の定義と特定方法，および指定したオブジェクトの生成に関わる部分グラフの抽出方法について説明する．

3.1　実行履歴

実行履歴とは，プログラムの実行中に行われた内容を実行順に記録したものである．実行履歴を記録するには実行履歴を取得したいプログラムに取得したい情報を記録するためのコードを埋め込み，コードを埋め込んだプログラムを実際に実行することで実行履歴を取得することができる．情報を取得するコードを埋め込む方法としてはソースコード中に埋め込む方法やバイトコード中に埋め込む方法がある．

本手法は対象となるプログラムの実行履歴を用いて，オブジェクト生成関係を特定している．利用者は対象となるプログラムを実行し，実行中に発生するイベントとイベントの詳細情報を実行履歴として取得する必要がある．

本手法で用いる実行履歴は以下のイベントで構成されている．

- コンストラクタ呼び出し (開始，終了)
- メソッド呼び出し (開始，終了)
- フィールドアクセス (書き込み，読み出し)
- 配列アクセス (書き込み，読み出し)

また，各イベントに対し，以下の情報を記録する．

全イベントに共通の情報

- イベントを実行したオブジェクトの ID とクラス情報
- イベントを実行したスレッドの ID，スレッド名
- 実行順序 (タイムスタンプ)

・対応するソースコードファイル名と行番号

コンストラクタ呼び出しに関する情報

・呼び出し先のオブジェクトの ID とクラス情報

・各引数の値情報

・発生した例外のオブジェクトの ID とクラス情報

・シグネチャ(コンストラクタ名，各引数の型，定義されたクラス)

メソッド呼び出しに関する情報

・呼び出し先のオブジェクトの ID とクラス情報

・各引数の値情報

・戻り値の値情報

・発生した例外のオブジェクトの ID とクラス情報

・シグネチャ(メソッド名，各引数の型，戻り値の型，定義されたクラス)

フィールドアクセスに関する情報

・対象フィールドを持つオブジェクトの ID とクラス情報

・読み書きした値情報

・シグネチャ(フィールド名，型，定義されたクラス)

配列アクセスに関する情報

・配列の ID と型

・インデックス

・読み書きした値情報

なお，クラス情報にはそのクラスのスーパークラスや実装しているインタフェースの情報を含んでいる．値情報では，プリミティブ型ならば型名とその値，オブジェクトならば ID とクラス情報を記録している．また，ID とはオブジェクトやスレッドを識別するためのも

のであり，スレッドやオブジェクトと ID は一対一で対応している．ID は実行履歴の記録時に自動的に割り振られる．

本手法において記録するイベントは対象となるプログラムで定義されているクラス中で発生するもののみとし，利用しているライブラリ中で発生するイベントは記録しない．たとえば，対象プログラムで定義されているクラスのオブジェクトから java.lang.String オブジェクトの split メソッドが呼び出された場合，記録されるイベントは split メソッドの呼び出しイベントだけであり，split メソッド中で発生するイベントは記録されない．ただし，コールバックが発生した場合はコールバックにより呼び出されたメソッド中に発生するイベントも記録する．たとえば，対象プログラムで定義されているクラスであり，かつ hashCode メソッドをオーバライドしているクラスのオブジェクトを引数として，java.util.HashSet オブジェクトの add メソッドが呼び出された場合，add メソッドの呼び出しからその終了までの間に，hashCode メソッドの呼び出しイベントと hashCode メソッド内で発生したイベントが記録される．

実行履歴上の区間を指定する際には区間の最初のイベントと最後のイベントのタイムスタンプを指定する．実行履歴はイベントをタイムスタンプ順に並べたイベント列であるため，実行履歴上の区間とは区間の最初のイベントが実行される直前から区間の最後のイベントが実行された直後までの時間区間を表す．

3.2 オブジェクト生成関係

オブジェクトがどのオブジェクトにより構築され，どのオブジェクトを生成に利用したかを表すため，オブジェクト生成関係を定義した．オブジェクト生成関係として TRIGGER と BASE の 2 種類を定義している．

TRIGGER オブジェクトを構築したことを表す

BASE オブジェクトの生成に利用されたことを表す

- オブジェクトに格納された
- オブジェクトに格納されたデータを取得するために利用された

オブジェクト A からオブジェクト B にオブジェクト生成関係 X が成立することを以下のように示す．

$$A \xrightarrow{X} B$$

$A \xrightarrow{TRIGGER} B$ の場合はオブジェクト A がオブジェクト B を構築したことを表し，$A \xrightarrow{BASE} B$ の場合はオブジェクト A がオブジェクト B の生成に利用されたことを表す．矢印の向きは，制御あるいはデータの流れを意味している．

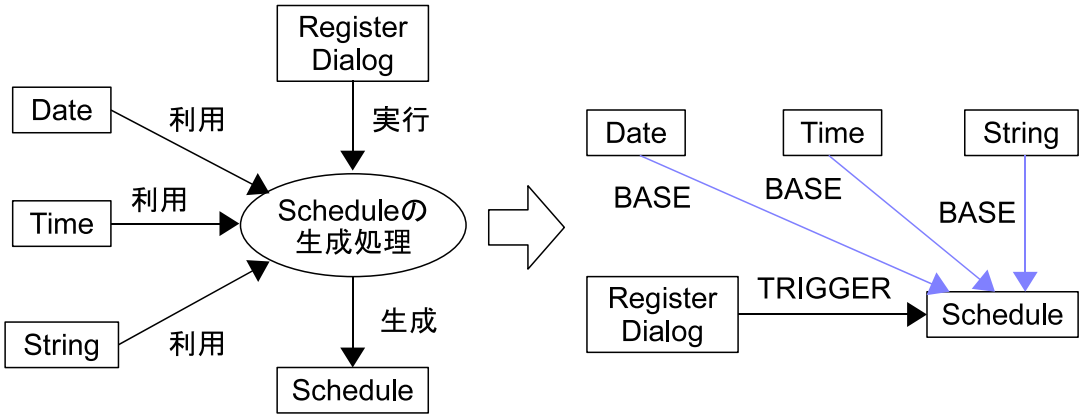

図 1: オブジェクト生成関係の例

図 1 にオブジェクト生成関係の例を示す．左の図で示すように予定登録ダイアログを表す RegisterDialog オブジェクトが予定を表す Schedule オブジェクトの生成処理を実行しており，Schedule オブジェクトの生成処理では日付を表す Date オブジェクトと時刻を表す Time オブジェクト，予定内容の文字列を表す String オブジェクトを利用している．この場合，オブジェクト生成関係は右の図で示すように RegisterDialog オブジェクトと Schedule オブジェクト間に TRIGGER が成立し，Date オブジェクト，Time オブジェクト，String オブジェクトと Schedule オブジェクトの間に BASE が成立する．

3.3 生成に関係するメソッド

オブジェクト生成関係を特定するために，オブジェクトの生成に関係するメソッドを定義する．

オブジェクトが構築されるときには，コンストラクタが呼び出されるため，コンストラクタ内で実行される処理はオブジェクトの生成に関係している．しかし，オブジェクトをある特定の状態にするには複数のメソッド実行が必要になるため，コンストラクタのみでなく，他のメソッドもオブジェクトの生成に関係していると考えることができる．

そこで次の 2 つのメソッド内で実行される処理はオブジェクトの生成に関係すると考える．

1. オブジェクトの状態を変更することを目的とするメソッド (状態変更メソッド)
2. コンストラクタや状態変更メソッドの実行中にのみ呼び出され，かつ呼び出し元のコンストラクタや状態変更メソッドと同一オブジェクト中に存在するメソッド (生成委譲メソッド)

本手法では，オブジェクトの状態を変更することを目的とするメソッドを状態変更メソッ

ドと呼び，この状態変更メソッドを生成に関係するメソッドの 1 つとした．メソッド内でオブジェクトの状態を変更したとしても，メソッド中のすべての処理が状態変更のために実行されているとは限らないため，状態変更のために実行されない処理を生成に関係するとしてしまうと誤ってオブジェクト生成関係を特定してしまう可能性がある．そのため，本手法では状態変更メソッドを特定し，状態変更メソッド内の処理はすべて生成に関係すると考えた．メソッドの目的はその名前で表現されることが多いことから，メソッド名の接頭辞が表 1 に示す 8 つの接頭辞のうちいずれかに該当する場合，そのメソッドを状態変更メソッドと判断する．この 8 つの接頭辞を選んだ理由も表 1 で示す．

また，本手法ではコンストラクタや状態変更メソッドの実行中にのみ呼び出され，かつ呼び出し元のコンストラクタや状態変更メソッドと同一オブジェクト中に存在するメソッドを生成委譲メソッドと呼び，この生成委譲メソッドも生成に関係するメソッドとした．コンストラクタや状態変更メソッドでは処理の一部を別メソッドとして同一オブジェクト中に持っている可能性がある．もしこの別メソッドが状態変更メソッドと判断されなかった場合，この別メソッド中の処理は生成に関係していないと判断されてしまう．しかし，コンストラクタや状態変更メソッドの実行中に呼び出される同一オブジェクト中の全メソッドを生成に関係するメソッドとしてしまうと，実際は生成に関係のないメソッドまで，生成に関係するメソッドに含めてしまう可能性がある．そこで，本手法ではメソッドがどのメソッドから呼び出されたかを解析し，生成に関係しないメソッドからも利用されるメソッドを除外することで，生成委譲メソッドを特定し，生成委譲メソッド内の処理はすべて生成に関係すると考えた．本手法では実行履歴中に含まれるコンストラクタ呼び出しイベントとメソッド呼び出しイベントを解析することでメソッドがどのメソッドから呼び出されていたかを特定する．

表 1: 状態変更メソッドと判断するための接頭辞

接頭辞	理由
set	フィールドに値を書き込む場合に利用
add	java.util.Collection などで要素を追加する場合に利用
put	java.util.Map でマッピングを追加する場合に利用
push	java.util.Stack で要素を追加する場合に利用
offer	java.util.Queue で要素を追加する場合に利用
append	java.lang.StringBuilder などで文字を追加する場合に利用
insert	java.lang.StringBuilder などでデータを挿入する場合に利用
replace	java.lang.StringBuilder などでデータを置き換える場合に利用

3.4 クラスフィールドとクラスメソッドの扱い

フィールドとメソッドは各オブジェクトが所持しているが，クラスフィールドとクラスメソッドはクラスごとに用意され，特定のオブジェクトには所属しない．そこで，本手法ではクラスフィールドとクラスメソッドのみを持つオブジェクトがクラスごとに 1 つ存在するとし，このオブジェクトをそのクラスの静的オブジェクトと呼ぶ．

静的オブジェクトも通常オブジェクトと同様にオブジェクト生成関係の特定を行うが，静的オブジェクトに対する TRIGGER の特定は行わない．Java におけるクラスフィールドは，クラスに定義された初期化処理の実行時に初期化されるため，クラスの初期化処理が実行されたときにその静的オブジェクトが構築されたと判断する．しかし，クラスの初期化処理はそのクラスを利用する際に Java の仮想マシンによって自動的に実行されるため，開発者は静的オブジェクトの構築を意識する必要はなく，生成に利用されたオブジェクトのみを把握すればよい．そのため，静的オブジェクトに対する TRIGGER は特定せずに，静的オブジェクトに対する BASE のみを特定する．

3.5 オブジェクト生成関係の特定

実行履歴に記録されたイベントを実行された順に解析していくことで，オブジェクト生成関係の特定を行う．開発者が興味のある実行履歴上の区間を指定したとき，解析の対象となるイベントは実行履歴上の最初のイベントから指定区間の最後のイベントまでである．解析は以下の 2 つの部分に分かれる．

1. 実行履歴上の最初のイベントから指定区間の直前のイベントまでの解析
2. 指定区間内のイベントの解析

まず，指定区間の開始よりも前に存在しているオブジェクトを特定するために最初のイベントから指定区間の直前のイベントまでを解析し，これらのイベントで使用されているオブジェクトを指定区間の開始より前から存在しているオブジェクトとする．その後，指定区間内のイベントを順に解析していくことでオブジェクト生成関係の特定を行う．

3.5.1 既知のオブジェクト集合

オブジェクト生成関係を特定する際にはイベントで使用されているオブジェクトが初めて登場したのか，それまでのイベントですでに登場したのかを判断する必要がある．そのため，本手法では既知のオブジェクト集合を用意している．各イベントで使用されているオブジェクトが既知のオブジェクト集合に含まれていれば，それまでのイベントで登場している既知

のオブジェクトであると判断し，含まれていなければ，そのイベントで初登場した未知のオブジェクトであると判断する．

最初のイベントから指定区間の直前のイベントまでの解析では，イベントで使用されているオブジェクトを既知のオブジェクト集合に追加しておく．また，指定区間内のイベントを解析する際にも未知のオブジェクトが登場するごとに，その未知のオブジェクトを既存のオブジェクト集合に追加する．

3.5.2 TRIGGER の特定

通常，オブジェクトが構築されるときには，コンストラクタが呼び出されるため，コンストラクタ呼び出しイベントを調べることで TRIGGER を特定できる，しかし，本手法ではライブラリ内部の動作を実行履歴に記録しないため，コンストラクタ呼び出しを記録できないオブジェクトが存在する．そこで本手法ではオブジェクトは初めて登場した際に構築されたとして TRIGGER の特定を行う．

実行履歴に記録されている内容によっては，TRIGGER を特定することが困難な場合や構築したオブジェクトが不明な場合が存在する．たとえば，マウスイベントが通知された場合，マウスイベントの生成はシステム内部で行われるため，どのオブジェクトが構築したかが分からないうえに，通知元のオブジェクトも記録することができない．そのため，マウスイベントがどのオブジェクトにより構築されたかを特定することは困難である．そこで，本手法では UNKNOWN オブジェクトという形式上のオブジェクトを用意し，TRIGGER を特定することが困難な場合や構築したオブジェクトが不明な場合は，UNKNOWN オブジェクトがオブジェクトを構築したと判断する．UNKNOWN オブジェクトは特定の際に 1 つしか用意しない．

ここから各イベントごとにおける TRIGGER の特定方法を説明する．

全イベント共通に行う TRIGGER の特定

通常，イベントを実行するオブジェクトはコンストラクタ呼び出しやメソッド呼び出しの呼び出し先のオブジェクトとして初登場しており，イベントを実行するオブジェクトとして初めて登場することはない．そのため，イベントを実行するオブジェクトが未知のオブジェクトである場合，そのオブジェクトに対する TRIGGER を特定することは困難である．したがって，イベントを実行したオブジェクトの TRIGGER を以下のように特定する．

「イベントを実行したオブジェクトが未知の場合」

UNKNOWN オブジェクト $\xrightarrow{TRIGGER}$ イベントを実行したオブジェクト

コンストラクタ呼び出しイベントにおける TRIGGER の特定

オブジェクトが構築されるときには，コンストラクタが呼び出されるため，コンストラクタの呼び出し先のオブジェクトが未知のオブジェクトである場合，そのオブジェクトは呼び出し元のオブジェクトによって構築されたはずである．したがって，呼び出し先のオブジェクトの TRIGGER を以下のように特定する．

「コンストラクタの呼び出し先のオブジェクトが未知の場合」

呼び出し元のオブジェクト $\xrightarrow{TRIGGER}$ 呼び出し先のオブジェクト

また，コンストラクタ呼び出しの引数が未知のオブジェクトである場合，そのオブジェクトはコンストラクタを呼び出す際に呼び出し元のオブジェクトによって構築されたと考える．したがって，引数のオブジェクトの TRIGGER を以下のように特定する．

「コンストラクタの引数のオブジェクトが未知の場合」

呼び出し元のオブジェクト $\xrightarrow{TRIGGER}$ 引数のオブジェクト

また，コンストラクタ実行中に投げられた例外が未知のオブジェクトである場合，呼び出し元のオブジェクトがそのコンストラクタを呼び出すことにより，例外のオブジェクトを構築したと考える．したがって，例外のオブジェクトの TRIGGER を以下のように特定する．

「コンストラクタ実行中に投げられた例外のオブジェクトが未知の場合」

呼び出し元のオブジェクト $\xrightarrow{TRIGGER}$ 例外のオブジェクト

メソッド呼び出しイベントにおける TRIGGER の特定

メソッド呼び出し時の呼び出し先のオブジェクトや引数のオブジェクトが未知のオブジェクトである場合，そのオブジェクトはメソッドを呼び出す際に呼び出し元のオブジェクトによって構築されたと考える．したがって，呼び出し先のオブジェクトと引数のオブジェクトの TRIGGER を以下のように特定する．

「メソッドの呼び出し先のオブジェクトが未知の場合」

呼び出し元のオブジェクト $\xrightarrow{TRIGGER}$ 呼び出し先のオブジェクト

「メソッドの引数のオブジェクトが未知の場合」

呼び出し元のオブジェクト $\xrightarrow{TRIGGER}$ 引数のオブジェクト

また，メソッドの戻り値やメソッド実行中に投げられた例外が未知のオブジェクトである

場合，呼び出し元のオブジェクトがそのメソッドを呼び出すことにより，戻り値や例外のオブジェクトを構築したと考える．したがって，戻り値のオブジェクトと例外のオブジェクトの TRIGGER を以下のように特定する．

「メソッドの戻り値のオブジェクトが未知の場合」

呼び出し元のオブジェクト $\xrightarrow{TRIGGER}$ 戻り値のオブジェクト

「メソッド実行中に投げられた例外のオブジェクトが未知の場合」

呼び出し元のオブジェクト $\xrightarrow{TRIGGER}$ 例外のオブジェクト

フィールドアクセスイベントにおける TRIGGER の特定

アクセスしたフィールドを持つオブジェクトが未知のオブジェクトである場合，そのオブジェクトはフィールドにアクセスしたオブジェクトが構築したと考える．したがって，フィールドを持つオブジェクトの TRIGGER を以下のように特定する．

「アクセスしたフィールドを持つオブジェクトが未知の場合」

アクセスしたオブジェクト $\xrightarrow{TRIGGER}$ フィールドを持つオブジェクト

フィールドへ値を書き込んだ際に，書き込んだ値が未知のオブジェクトである場合，そのオブジェクトはフィールドにアクセスしたオブジェクトが構築したと考える．したがって，フィールドへ書き込まれたオブジェクトの TRIGGER を以下のように特定する．

「フィールドに書き込まれたオブジェクトが未知の場合」

アクセスしたオブジェクト $\xrightarrow{TRIGGER}$ 書き込まれたオブジェクト

フィールドから値を読み出した際に，読み出した値が未知のオブジェクトである場合，そのオブジェクトはフィールドを持つオブジェクトが構築し，フィールドに書き込んでいたと考える．したがって，フィールドから読み出したオブジェクトの TRIGGER を以下のように特定する．

「フィールドから読み出したオブジェクトが未知の場合」

フィールドを持つオブジェクト $\xrightarrow{TRIGGER}$ 読み出したオブジェクト

配列アクセスイベントにおける TRIGGER の特定

アクセスした配列が未知のオブジェクトである場合，その配列は配列にアクセスしたオブジェクトが構築したと考える．したがって，配列の TRIGGER を以下のように特定する．

「アクセスした配列が未知の場合」

アクセスしたオブジェクト $\xrightarrow{TRIGGER}$ 配列

配列へ値を書き込んだ際に，書き込まれた値が未知のオブジェクトである場合，そのオブジェクトは配列にアクセスしたオブジェクトが構築したと考える．したがって，配列へ書き込まれたオブジェクトの TRIGGER を以下のように特定する．

「配列に書き込まれたオブジェクトが未知の場合」

アクセスしたオブジェクト $\xrightarrow{TRIGGER}$ 書き込まれたオブジェクト

配列から値を読み出した際に，読み出された値が未知のオブジェクトである場合，そのオブジェクトは配列の構築と同時に構築され，配列に書き込まれていたと考える．したがって，配列から読み出されたオブジェクトの TRIGGER を以下のように特定する．

「配列から読み出されたオブジェクトが未知の場合」

配列を構築したオブジェクト $\xrightarrow{TRIGGER}$ 読み出したオブジェクト

3.5.3 BASE の特定

各イベントで利用されているオブジェクトが生成に利用されているかを判断することで BASE の特定を行う．BASE を特定する際には同一オブジェクト間の BASE を特定する場合がある．たとえば，オブジェクト A がフィールドに持つデータを利用して，オブジェクト A の別のフィールドのデータを生成した場合，$A \xrightarrow{BASE} A$ という BASE を特定することができる．しかし，自身の持つデータは常に利用可能であるため，本手法では同一オブジェクト間の BASE は省略する．

今後の説明のため，BASE を以下の 2 種類に分ける．

BASE-A 利用時にオブジェクトが保持し，利用後も参照可能

BASE-U 一時的に利用されるだけであり，利用後は参照不可

ここから各イベントにおける BASE の特定方法を説明する．

コンストラクタ呼び出しイベントにおける BASE の特定

コンストラクタは生成に関係しているため，コンストラクタの引数のオブジェクトは生成に利用されたはずであるが，引数として渡された時点ではどのように利用されたかは分からない．そのため，コンストラクタ内で実際に引数のオブジェクトが利用された際に BASE を

特定すべきである．しかし，ライブラリのように内部の処理が記録されていない場合はオブジェクトがコンストラクタの引数として渡された時点で BASE を特定する必要がある．したがって，ライブラリのコンストラクタが呼び出された場合は，呼び出し先のオブジェクトの BASE を以下のように特定する．

「ライブラリのコンストラクタが呼び出された場合」

引数のオブジェクト $\xrightarrow{BASE-A}$ 呼び出し先のオブジェクト

コンストラクタ内ではフィールドの初期化が行われることが多いため，利用後も引数のオブジェクトを参照可能である可能性が高いと考え，BASE-A として特定する．

メソッド呼び出しイベントにおける BASE の特定

3.5.2 項のメソッド呼び出しイベントにおける TRIGGER の特定で述べたようにメソッドの戻り値が未知のオブジェクトであった場合，メソッドを呼び出すことで戻り値のオブジェクトを構築したと考えた．メソッド中では呼び出し先のオブジェクトと引数のオブジェクトを用いて処理を行っているため，戻り値のオブジェクトの生成にこれらのオブジェクトが利用されたと考えた．したがって，メソッドの戻り値が未知のオブジェクトであった場合は，戻り値のオブジェクトの BASE を以下のように特定する．

「メソッドの戻り値が未知のオブジェクトの場合」

呼び出し先のオブジェクト $\xrightarrow{BASE-U}$ 戻り値のオブジェクト

引数のオブジェクト $\xrightarrow{BASE-U}$ 戻り値のオブジェクト

呼び出し先のオブジェクトや引数のオブジェクトは戻り値のオブジェクトの生成に直接利用されたかどうかは判断できないため，BASE-U として特定する．

コンストラクタや生成に関係するメソッド中で戻り値が存在し，かつ戻り値が未知のオブジェクトでないメソッドが呼び出された場合，オブジェクトの生成に利用するために戻り値を取得したと考えることができる．そのため，戻り値を取得するために利用した呼び出し先のオブジェクトや引数のオブジェクトもオブジェクトを生成するために利用したと考えることができる．したがって，コンストラクタや生成に関係するメソッド中で戻り値が存在し，かつ戻り値が未知のオブジェクトでないメソッドが呼び出された場合は，呼び出し元のオブジェクトの BASE を以下のように特定する．

「コンストラクタや生成に関係するメソッド中で戻り値が存在し，
かつ戻り値が未知のオブジェクトでないメソッドが呼び出された場合」

呼び出し先のオブジェクト $\xrightarrow{BASE-U}$ 呼び出し元のオブジェクト

引数のオブジェクト $\xrightarrow{BASE-U}$ 呼び出し元のオブジェクト

呼び出し先のオブジェクトと引数のオブジェクトは戻り値を取得するために利用しているので，BASE-U として特定する．戻り値が未知のオブジェクトである場合を除外しているが，戻り値が未知のオブジェクトである場合は呼び出し先のオブジェクトや引数のオブジェクトは戻り値のオブジェクトを生成するために利用したのであり，呼び出し元のオブジェクトの生成に利用したわけではないと考えたためである．

コンストラクタや生成に関係するメソッドから戻り値がオブジェクトであるメソッドが呼び出された場合，戻り値のオブジェクトを生成に利用していると考えることができるが，戻り値として取得した時点ではどのように利用されたかは分からない．そのため，実際に戻り値のオブジェクトが利用された際に BASE を特定すべきである．しかし，そのメソッドがライブラリから呼び出されている場合は戻り値のオブジェクトの利用が記録されないため，戻り値として渡された時点で BASE を特定する必要がある．したがって，ライブラリのコンストラクタや生成に関係するメソッドから戻り値がオブジェクトであるメソッドが呼び出された場合は，呼び出し元のオブジェクトの BASE を以下のように特定する．

「ライブラリのコンストラクタや生成に関係するメソッドから
戻り値がオブジェクトであるメソッドが呼び出された場合」

戻り値のオブジェクト $\xrightarrow{BASE-U}$ 呼び出し元のオブジェクト

生成に関係するメソッドの引数のオブジェクトはオブジェクトの生成に利用されたはずであるが，引数として渡された時点ではどのように利用されたかは分からない．そのため，メソッド内で実際に引数のオブジェクトが利用された際に BASE を特定すべきである．しかし，ライブラリのように内部の処理が記録されていない場合はオブジェクトがメソッドの引数として渡された時点で BASE を特定する必要がある．したがって，ライブラリの生成に関わるメソッドが呼び出された場合は，呼び出し先のオブジェクトの BASE を以下のように特定する．

「ライブラリの生成に関わるメソッドが呼び出された場合」

引数のオブジェクト $\xrightarrow{BASE-A}$ 呼び出し先のオブジェクト

生成に関わるメソッドはコンストラクタと同等に考えているため，利用後も引数のオブジェ

クトを参照可能である可能性が高いと考え，BASE-A として特定する．

フィールドアクセスイベントにおける BASE の特定

フィールドへオブジェクトが書き込まれた場合，書き込まれたオブジェクトを利用してフィールドを持つオブジェクトの状態を変更したと考えることができる．したがって，フィールドへオブジェクトが書き込まれた場合は，フィールドを持つオブジェクトの BASE を以下のように特定する．

「フィールドへオブジェクトが書き込まれた場合」

書き込まれたオブジェクト $\xrightarrow{BASE-A}$ フィールドを持つオブジェクト

コンストラクタや生成に関係するメソッド中でフィールドからデータを読み出した場合，オブジェクトの生成に利用するためにデータを読み出したと考えることができる．そのため，オブジェクト生成のためにそのフィールドを持つオブジェクトを利用したと考えることができる．したがって，コンストラクタや生成に関係するメソッド中でフィールドからデータを読み出した場合は，フィールドへアクセスしたオブジェクトの BASE を以下のように特定する．

「コンストラクタや生成に関係するメソッド中で
フィールドからデータを読み出した場合」

フィールドを持つオブジェクト $\xrightarrow{BASE-U}$ アクセスしたオブジェクト

フィールドからデータを読み出しているので，BASE-U として特定する．読み出されたデータがオブジェクトの場合，そのオブジェクトを利用したと考えることができるが，読み出した時点ではどのように利用されたかは分からない．そのため，実際にフィールドから読み出されたオブジェクトが利用された際に BASE の特定を行う．

配列アクセスイベントにおける BASE の特定

3.5.2 項の配列アクセスイベントにおける TRIGGER の特定で述べたように配列から読み出されたデータが未知のオブジェクトであった場合，読み出されたオブジェクトは配列と同時に構築されたと考えた．そのため，配列の生成に利用されたオブジェクトが読み出されたオブジェクトの生成にも利用されたと考えるべきであるが，配列への書き込むものは除外すべきである．また，読み出されたオブジェクトは既に配列に書き込まれていたため，読み出されたオブジェクトは配列の生成に利用されたはずである．したがって，配列から読み出されたデータが未知のオブジェクトであった場合は，配列から読み出されたオブジェクトと配列の BASE を以下のように特定する．

「配列から読み出されたデータが未知のオブジェクトであった場合」

配列の生成に利用されたオブジェクト $\xrightarrow{BASE-U}$ 読み出されたオブジェクト

($BASE-U$ として特定されたもののみ)

読み出されたオブジェクト $\xrightarrow{BASE-A}$ 配列

配列へオブジェクトが書き込まれた場合，書き込まれたオブジェクトを利用して配列の状態を変更したと考えることができる．したがって，配列へオブジェクトが書き込まれた場合は，配列の BASE を以下のように特定する．

「配列へオブジェクトが書き込まれた場合」

書き込まれたオブジェクト $\xrightarrow{BASE-A}$ 配列

コンストラクタや生成に関係するメソッド中で配列からデータを読み出した場合，オブジェクトの生成に利用するためにデータを読み出したと考えることができる．そのため，オブジェクト生成のためにその配列を利用したことになる．したがって，コンストラクタや生成に関係するメソッド中で配列からデータを読み出した場合は，配列にアクセスするオブジェクトの BASE を以下のように特定する．

「コンストラクタや生成に関係するメソッド中で
配列からデータを読み出した場合」

配列 $\xrightarrow{BASE-U}$ アクセスしたオブジェクト

配列からデータを読み出しているので，BASE-U として特定する．読み出されたデータがオブジェクトの場合，配列から読み出されたオブジェクトを利用したと考えることができるが，読み出した時点ではどのように利用されたかは分からない．そのため，実際に配列から読み出されたオブジェクトが利用された際に BASE の特定を行う．

3.6 指定オブジェクトの生成に関係する部分グラフの抽出

ある特定のオブジェクトの生成に注目するため，指定区間全体のオブジェクト生成関係から指定したオブジェクトの生成に関係する部分を抽出する．

特定のオブジェクトの生成に関係する部分を抽出するには，対象オブジェクトを生成するために必要なオブジェクトのみを含む部分グラフを抽出する必要がある．対象オブジェクトを生成するために必要なオブジェクトを特定するには，オブジェクトの利用を表す BASE を対象オブジェクトから逆にたどることで特定できる．しかし，BASE をたどるときは単にたどることのできる BASE を全部たどってはいけない．なぜならば，BASE には対象オブ

ジェクトの生成に関係した後に特定されたものも含まれているからである．すなわち，たどる BASE はオブジェクトが対象オブジェクトの生成に関係する以前に特定された BASE のみである．これは関係を特定した際に，特定したイベントのタイムスタンプを記録しておき，タイムスタンプを比較することで対象オブジェクトの生成に関係する以前に特定された BASE を識別できる．

しかし，対象オブジェクトの生成に関係する以前の BASE をたどるだけでは問題がある．オブジェクト状態はオブジェクトが管理しているデータの内容により決まるが，オブジェクトが他のオブジェクトを保持している場合，保持しているオブジェクトの状態が変化すると保持している側のオブジェクトの状態も変化したとみなすべきである．たとえば，オブジェクト A のフィールドにオブジェクト B を書き込み，書き込んだ後にオブジェクト C をオブジェクト B のフィールドに書き込んだとする．このとき，BASE は $B \xrightarrow{BASE} A$ と $C \xrightarrow{BASE} B$ の 2 つが特定される．$B \xrightarrow{BASE} A$ を特定した際のイベントのタイムスタンプを x，$C \xrightarrow{BASE} B$ を特定した際のイベントのタイムスタンプを y とすると，$x < y$ が成立する．オブジェクト A の生成に関係する部分を抽出するとき，オブジェクト B と C はオブジェクトの生成に必要なオブジェクトと判断すべきである．しかし，対象オブジェクトの生成に関係する以前の BASE をたどることでオブジェクト A の生成に必要なオブジェクトを特定すると，$x < y$ であるためオブジェクト A から $B \xrightarrow{BASE} A$ をたどった後に $C \xrightarrow{BASE} B$ をたどることはないため，オブジェクト C はオブジェクト A の生成に必要ないと判断されてしまう．そのため，利用後も参照可能であると考えている BASE-A が対象オブジェクトから連続でたどることができる場合，特定した際のイベントのタイムスタンプに関わらずたどるべきである．

BASE をたどる際の基準を以下にまとめる[1]．

- 現在までに BASE-A しかたどっていない場合
  - BASE-A は無条件にたどる
  - BASE-U は直前にたどった BASE のタイムスタンプよりも前のタイムスタンプを持つもののみをたどる
- 現在までに BASE-U を 1 回でもたどった場合
  - BASE-A，BASE-U ともに直前にたどった BASE のタイムスタンプよりも前のタイムスタンプを持つもののみをたどる

この基準で BASE をたどっていき，たどった経路上の BASE とオブジェクトに加え，そのオブジェクトに対する TRIGGER とオブジェクトを残す．

[1]注：付録 B を参照

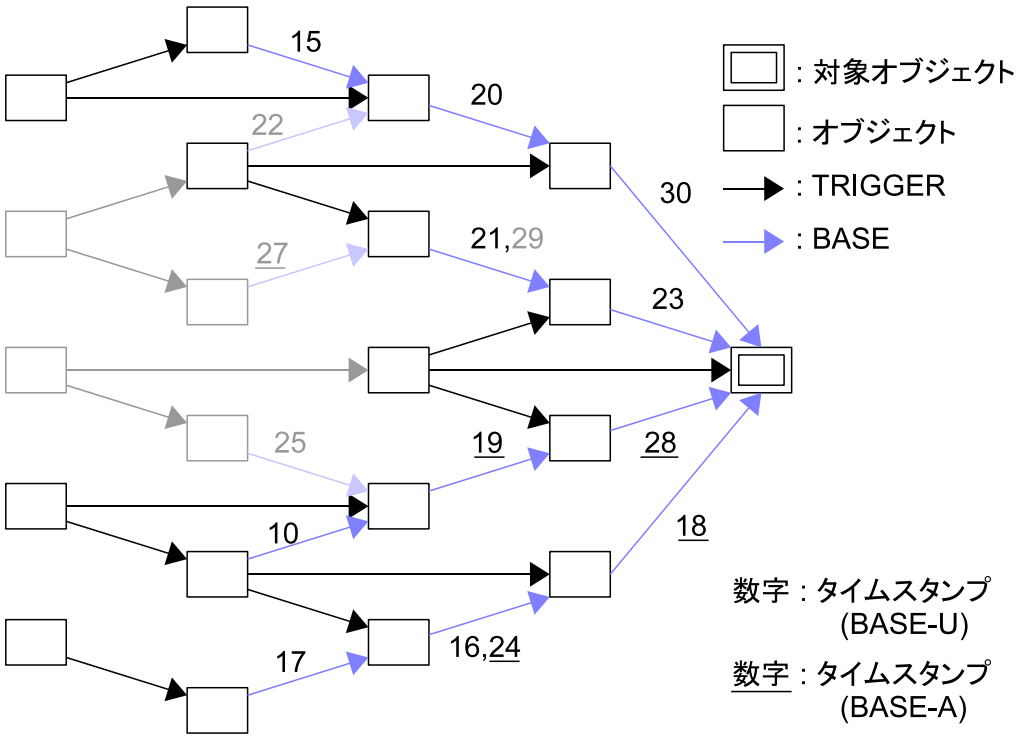

図 2: 対象オブジェクトの生成に関係する部分の抽出例

図 2 に対象オブジェクトの生成に関係する部分の抽出例を示す．薄く表示されている部分が除去する部分である．

3.6.1 BASE をたどる際の追加基準

指定区間全体のオブジェクト生成関係から対象オブジェクトの生成に関係する部分を抽出した結果のグラフが巨大なサイズになることを避けるため，以下の基準をオプションとして自由に追加可能にする．

基本的なクラスのオブジェクトに到達したらその先の BASE はたどらない

文字列や数値を表すオブジェクトは生成が容易であり，その内容から用途を予測しやすいのではないかと考え，基本的なクラスのオブジェクトに到達したらその先の BASE はたどらないという基準を用意した．今回は表 2 に示すクラスを基本的なクラスと考えた．

GUI 関連のクラスのオブジェクトに到達したらその先の BASE はたどらない

GUI を構成するオブジェクトからデータを取得する場合，どのオブジェクトから取得したかが分かればよく，そのオブジェクトの構成まで知る必要がない場合がある．そこで GUI

関連のクラスのオブジェクトに到達したらその先の BASE はたどらないという基準を用意した．以下のいずれかの条件を満たすクラスを GUI 関連のクラスと判断する．

・ 完全修飾クラス名に”.gui.”を含む

・ java.awt.Component クラスを継承している

オブジェクトの数が指定したサイズを超えた時点で終了する

この基準では BASE をたどることで到達可能なオブジェクトの最短経路の長さが短いものから順に加えていき，オブジェクト数が指定したサイズを超えた時点でそれよりも最短経路の長さが長いものは省略する．そして，残したオブジェクトのみで構成できる経路のみを残す．

表 2: 基本的なクラス

java.lang.Boolean	java.lang.Byte
java.lang.Charactor	java.lang.Double
java.lang.Float	java.lang.Integer
java.lang.Long	java.lang.Short
java.lang.Class	java.lang.String

4　実装

提案手法を実装したツールを作成した．本ツールでは実行可能な Java プログラムを解析対象とし，以下の手順で解析を行う．

1. 解析対象のプログラムを実行し，実行履歴を記録する．
2. 実行履歴を読み込む．
3. オブジェクト生成関係を抽出する区間を指定し，実行する．
4. 生成したグラフへの操作を行う．(任意)
   - オブジェクトを指定し，そのオブジェクトの生成に関係する部分グラフを抽出する．
   - 集約を行う．
5. 結果のグラフを表示する．

本ツールでは利用者が区間やオブジェクトを指定する際の支援機能や提示する有向グラフの集約機能などを実装している．

4.1　ツールの構成

ツールの構成を図 3，スクリーンショットを図 4 に示す．

本ツールは我々の研究グループで開発したツールである Amida[1] に含まれる Amida-Agent というトレーサを用いている．Amida-Agent 側では実行履歴の記録と読み込みを行い，本ツールでは Amida-Agent が読み込んだ実行履歴の内容をイベント情報として取得し，解析することでオブジェクト生成関係を特定している．本ツールはメソッド情報の解析と生成関係の特定，関係部分の抽出，生成関係の表示の 5 つで構成される．

実行履歴の記録

Amida-Agent を用いることで実行履歴をファイルに記録している．Amida-Agent を用いることで 3.1 節で挙げた情報を記録できる．また，表 3 に示すクラスの toString メソッドは，人間がオブジェクトの状態を理解するのに有用な文字列を返すため，これらのクラスを継承しているクラスのオブジェクトについては，そのオブジェクトの toString メソッドの戻り値を記録可能である．

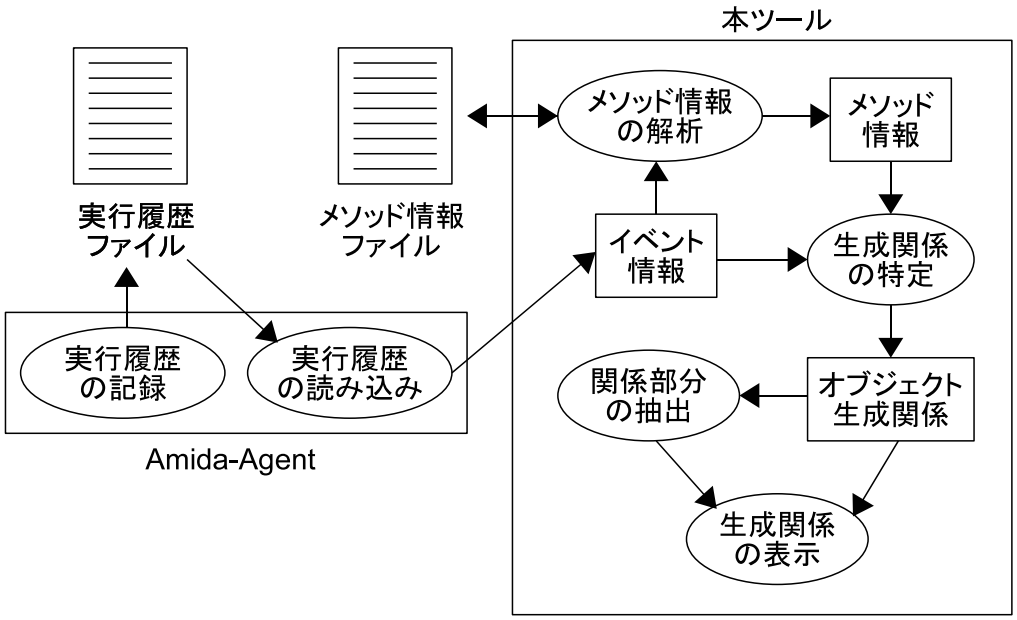

図 3: ツールの構成

実行履歴の読み込み

Amida-Agent により記録する実行履歴は複数のバイナリファイルとテキストファイルで構成されるため，そのまま扱うことは困難である．そのため，Amida-Agent は実行履歴を読み込み，解析可能なデータ構造に変換することができる．

本ツールは Amida-Agent が実行履歴を読み込むことで生成したイベント情報を取得して，解析を行う．

メソッド情報の解析

メソッド情報の解析では 3.3 節で説明した生成に関係するメソッドの判定のため，メソッド呼び出し関係を特定し，また，4.2.1 項で説明する区間指定の支援のため，実行履歴中に記録されている各メソッド実行の開始と終了のタイムスタンプの一覧を作成する．実行履歴から読み込んだイベント情報のうち，メソッド呼び出しイベントとコンストラクタ呼び出し

表 3: toString の値が取得可能なクラス

java.lang.Number	java.lang.Boolean
java.lang.Character	java.lang.String
java.lang.Class	java.lang.Enum
java.lang.Throwable	java.util.Date

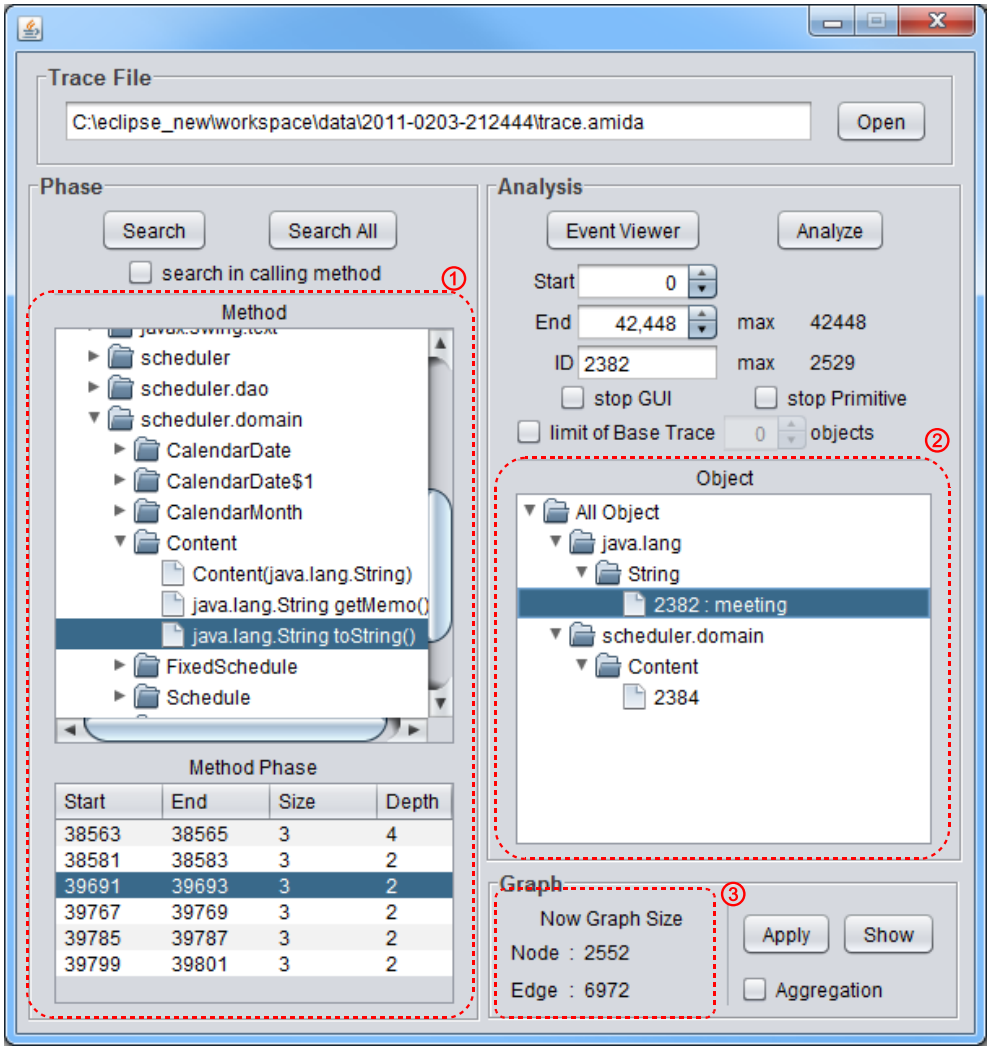

図 4: ツールのスクリーンショット

イベントをチェックしていくことで，メソッドの呼び出し関係と各メソッド実行の開始と終了のタイムスタンプを調べている．

解析後，各メソッド実行の開始と終了のタイムスタンプは図 4 の①の部分で示すように提示される．

この部分における解析結果は実行履歴に対して一意に決まるため，結果をファイルに保存しておくことにより，同じ実行履歴に対して生成関係の特定を行う際にはファイルから結果を読み込むだけでよく，次回以降の解析時間を短縮できる．

生成関係の特定

実行履歴から読み込んだイベント情報とメソッド情報の解析により取得したメソッド呼び

出し関係を用いて，利用者が指定した区間におけるオブジェクト生成関係の特定を行う．特定方法は 3.5 節で説明した通りである．

関係部分の抽出

特定したオブジェクト生成関係から，利用者が指定したオブジェクトの生成に関係する部分を抽出する．抽出方法は 3.6 節で説明した通りである．

また，4.2.2 項で説明するオブジェクトの指定支援のため，オブジェクトの ID 一覧を生成することができる．利用者の選択に応じて実行履歴から読み込んだイベントをチェックしていき，イベントで利用されているオブジェクトの ID を図 4 の②の部分に示すように提示している．なお，表 3 に示したクラスを継承したオブジェクトについては toString の値を ID と一緒に提示する．これにより，利用者がオブジェクトの指定を行いやすくなると考えている．

生成関係の表示

表示するオブジェクト生成関係は指定区間全体のグラフか，指定オブジェクトの生成に関係する部分グラフである．どちらの場合でも 4.3 節で説明する集約を行うことが可能である．

本ツールでは図 4 の③の部分に現在のグラフの頂点数と辺の数を提示しておく．これにより，サイズが大きすぎるものを表示してしまうことを避けることができると考えている．

グラフの生成には Graphviz[9] を利用し，表示には本ツールのビューアを用いている．

4.2 入力支援

利用者が何も情報を持たずに実行履歴上の区間を指定したり，注目するオブジェクトを指定することは困難であるため，これらを指定するための機能を用意した．

4.2.1 区間の指定支援

利用者が区間を指定する際はメソッド実行に注目する可能性が高いと考えられる．なぜなら，メソッドの開始直前を区間の終了とすることでそのメソッドを実行した際における状態のオブジェクトの生成方法を知ることができるし，メソッドの開始から終了までを指定区間とすることでそのメソッド内で実行された生成処理を知ることができ，メソッド内の処理の理解に役立てることができるからである．

そのため，本ツールでは解析対象の実行履歴における各メソッド実行の開始と終了のタイムスタンプの一覧を提示する．これにより，利用者は注目したいメソッド実行に対応するタイムスタンプを知ることができ，区間の指定に役立つと考えている．

4.2.2 オブジェクトの指定支援

特定した生成関係から特定のオブジェクトの生成に関係する部分を抽出する際には，開発者はオブジェクトの ID を指定する必要がある．しかし，オブジェクトの ID は自動的に割り振られているため，利用者が注目したいオブジェクトの ID を知ることは困難である．また，オブジェクトの一覧を提示したとしても，同一クラスのオブジェクトが大量に生成されている場合，注目したいオブジェクトを選択することが困難である．4.2.1 項で述べたように利用者はメソッド実行に注目する可能性が高いことから，本ツールでは指定したメソッド実行中に利用されたオブジェクトのリストを提示する．これにより，注目したいメソッド実行で利用されたオブジェクトの ID を知ることができ，オブジェクトの指定に役立つと考えている．

本ツールでは以下のオブジェクトのリストを提示することが可能である．

- 実行履歴中で利用される全オブジェクト
- 指定したメソッド実行中で利用されたオブジェクト
  - 指定したメソッド中で利用するもののみ
  - 指定したメソッド実行中に呼び出されるメソッド内で利用するオブジェクトも含む

4.3 グラフの集約

提示するグラフには同様の生成関係を表す部分が複数含まれていると考えられる．そこで，これらを集約することで提示するグラフのサイズを抑えることができる．

集約の基準は以下の通りである．

- 対応するオブジェクトのクラスが等しい
- 対応するオブジェクト間の生成関係が等しい

集約例を図 5 に示す．この左の図では B(1) と B(2) はともに A からの TRIGGER を持っており，クラス C のオブジェクトへの TRIGGE を持っている．同様に C(1) と C(2) はともに A からの BASE を持っており，クラス B のオブジェクトからの TRIGGER を持っている．したがって，B(1) と B(2)，C(1) と C(2) は集約することができ，集約すると右の図のようになる．

頂点が集約されているオブジェクトを表す場合は枠を二重にし，集約されたオブジェクトの数を表記する．

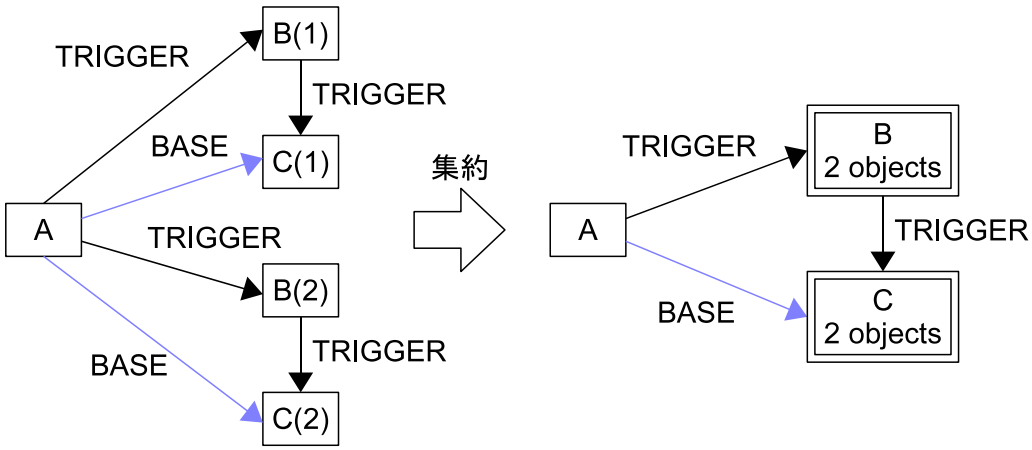

図 5: 集約例

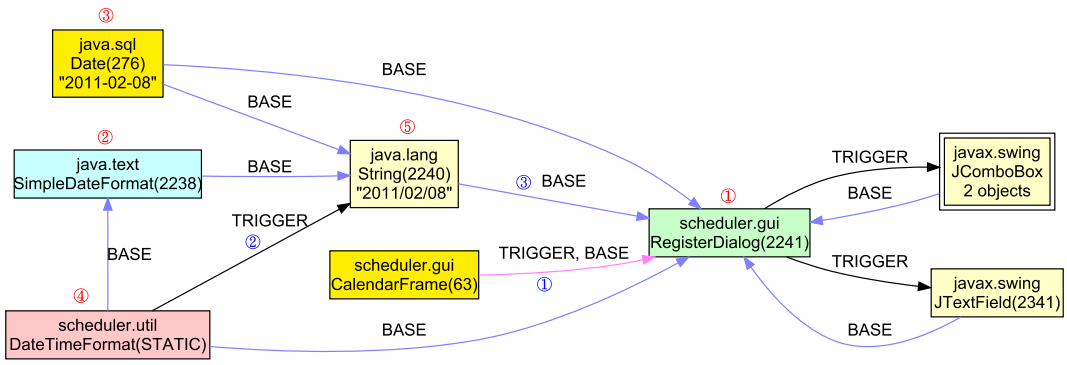

図 6: 表示されるグラフの例

4.4 表示グラフ

表示するグラフはオブジェクトを頂点，オブジェクト生成関係を辺とする有向グラフである．なお，同一オブジェクト間に複数のオブジェクト生成関係が成立している場合は一本にまとめている．図 6 に実際の表示例を示す．

頂点のラベルとして以下の情報を表示している．

- パッケージ名
- クラス名
- オブジェクト ID(集約されていない場合)
- 集約されている個数 (集約されている場合)
- toString の値 (集約されておらず，表 3 に示すクラスを継承している場合)

また，頂点は以下の基準で色分けしている．基準は上のほうが優先度が高い．それぞれ図 6 中の赤の番号に対応している．

1. 対象オブジェクト (オブジェクトの生成に関係する部分を抽出した場合)
2. 3.6.1 項で説明した BASE をたどる際の追加基準でオブジェクト数に制限を加えた場合に BASE をたどることを打ち切ったオブジェクト (オブジェクトの生成に関係する部分を抽出した場合)
3. 指定区間より前に存在したオブジェクト
4. 静的オブジェクト
5. その他のオブジェクト

辺にはオブジェクト生成関係名をつけ，以下の基準で色分けしている．それぞれ図 6 中の青の番号に対応している．

1. TRIGGER と BASE の両方を表すもの
2. TRIGGER を表すもの
3. BASE を表すもの

オブジェクトやオブジェクト生成関係の詳細な情報を知るためにグラフを表示するビューアでは頂点や辺にマウスを合わせることで詳細情報を記述したダイアログを表示するようにしている．表示するダイアログの例を図 7 に示す．このダイアログには以下の情報が含まれている．

オブジェクト詳細情報

- オブジェクト情報
  - 完全修飾クラス名
  - スーパークラスの完全修飾クラス名
  - 実装しているインタフェース一覧
  - ID(集約されていない場合)
  - 集約されている個数 (集約されている場合)
  - toString の値 (集約されておらず，表 3 に示すクラスを継承している場合)

・他のオブジェクトから自身へのオブジェクト生成関係一覧

・自身から他のオブジェクトへのオブジェクト生成関係一覧

・集約されているオブジェクトのオブジェクト詳細情報 (集約されている場合)

オブジェクト生成関係詳細情報

・両端のオブジェクト情報

・オブジェクト生成関係を特定した際のイベント情報 (集約されていない場合)

  – イベントを実行したスレッドの ID とスレッド名

  – イベントを実行したメソッドの呼び出しイベントに対応するソースコードファイル名と行番号

  – イベントを実行したメソッド実行の開始と終了のタイムスタンプ

  – イベントを実行したメソッドを呼び出したメソッドの呼び出し情報 (シグネチャ，レシーバオブジェクト，引数一覧，戻り値，例外)

  – イベントを実行したメソッドの呼び出し情報 (シグネチャ，レシーバオブジェクト，引数一覧，戻り値，例外)

  – 特定したイベントに対応するソースコードファイル名と行番号

  – 特定したイベントのタイムスタンプ (メソッド呼び出し，コンストラクタ呼び出しの場合，開始と終了)

  – 特定したイベントがコンストラクタ呼び出しの場合，コンストラクタ呼び出し情報 (シグネチャ，レシーバオブジェクト，引数一覧，例外)

  – 特定したイベントがメソッド呼び出しの場合，メソッド呼び出し情報 (シグネチャ，レシーバオブジェクト，引数一覧，戻り値，例外)

  – 特定したイベントがフィールドアクセスの場合，フィールドアクセス情報 (フィールドを持つオブジェクト，フィールド名，値)

  – 特定したイベントが配列アクセスの場合，配列アクセス情報 (配列，インデックス，値)

・集約されているオブジェクト生成関係のオブジェクト生成関係詳細情報 (集約されている場合)

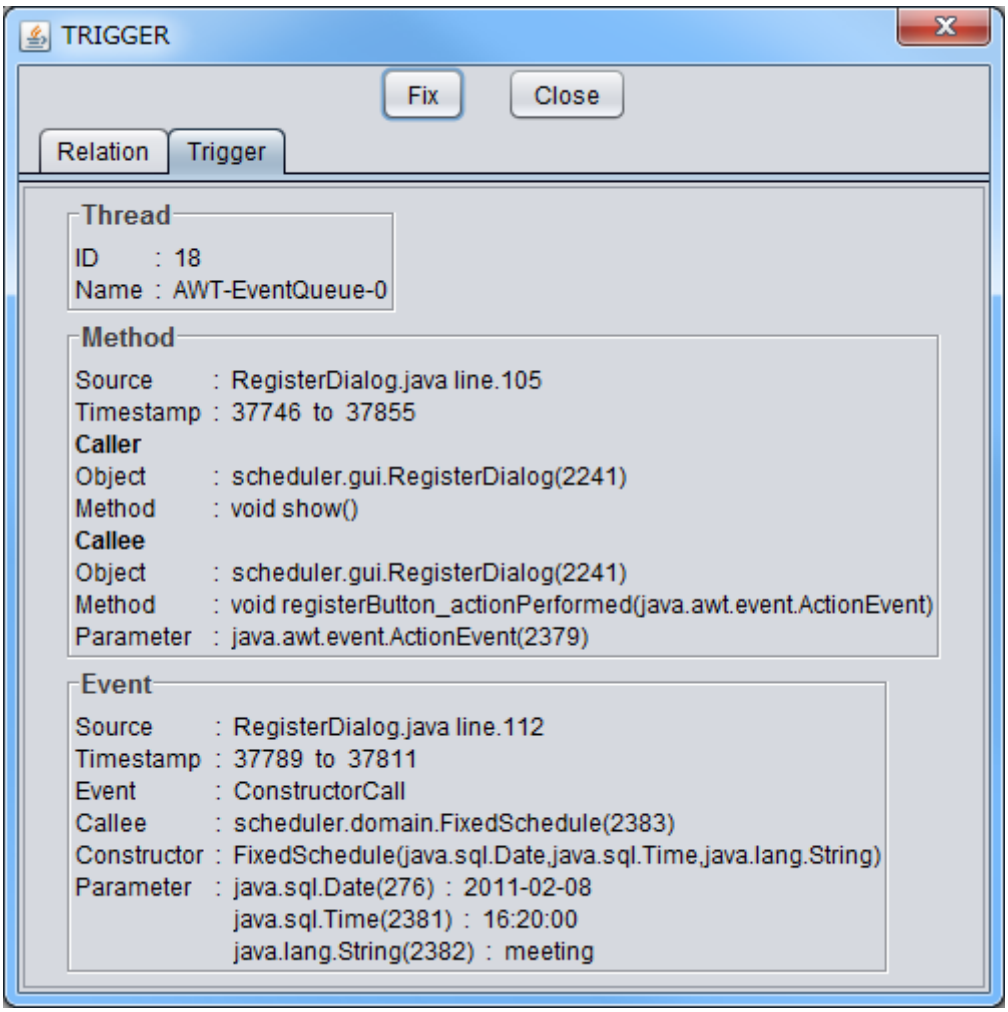
TRIGGER
Fix
Close
Relation
Trigger
Thread
ID : 18
Name : AWT-EventQueue-0
Method
Source : RegisterDialog.java line.105
Timestamp : 37746 to 37855
Caller
Object : scheduler.gui.RegisterDialog(2241)
Method : void show()
Callee
Object : scheduler.gui.RegisterDialog(2241)
Method : void registerButton_actionPerformed(java.awt.event.ActionEvent)
Parameter : java.awt.event.ActionEvent(2379)
Event
Source : RegisterDialog.java line.112
Timestamp : 37789 to 37811
Event : ConstructorCall
Callee : scheduler.domain.FixedSchedule(2383)
Constructor : FixedSchedule(java.sql.Date,java.sql.Time,java.lang.String)
Parameter : java.sql.Date(276) : 2011-02-08
java.sql.Time(2381) : 16:20:00
java.lang.String(2382) : meeting

図 7: 詳細ダイアログ

5　ケーススタディ

本章では 4 章で説明したツールを用いて実施したケーススタディについて説明する．

5.1 節と 5.2 節では解析対象として大阪大学基礎工学部情報科学科の 2 年生が演習で作成したライフゲームプログラムを用いる．このプログラムを用いた理由は著者がプログラムの内容を知らず，プログラムが小規模であり，ソースコードが利用可能であったためである．今回は L1 から L6 までの 6 つのライフゲームプログラムを利用した．各プログラムは同一の要求に基づいて作成されているが，作成者による独自の機能が追加されている．これらのライフゲームプログラムは GUI アプリケーションであり，起動するとウィンドウに盤面が表示される．この盤面は碁盤目状の格子で表され，各格子「セル」は「生きている」か,「死んでいる」状態のどちらかをとる．利用者はセルの状態を自由に操作することができ，盤面の更新を行うと現在の盤面の状態から次の盤面の状態を決定し，表示する．他にも履歴を用いて盤面を 1 つ前の状態に戻す巻き戻し機能や，盤面の状態をファイルに保存する機能，ファイルから盤面の状態を読み込む機能などがある．ライフゲームの仕様は付録に示す．

また，5.3 節では解析対象としてライフゲームのほかに Ant[2]，ANTLR[3]，DaCapo[6] を用いた．

5.1　実行シナリオの分析

本節ではライフゲームプログラム L4 を分析した結果を示す．

図 8 は L4 を起動し，盤面の状態を保存しているファイルを開いて，終了するというシナリオを実行した際に記録した実行履歴を解析し，取得したオブジェクト生成関係の一部である．この図では CellIO オブジェクトから Cell への TRIGGER が成立していることから，CellIO オブジェクトが Cell オブジェクトを構築していることが分かる．図 9 に示す CellIO オブジェクトから String オブジェクトへの TRIGGER の詳細情報を確認すると，RandomAccessFile オブジェクトの readLine メソッドを呼び出すことで String オブジェクトを構築したことが分かり，図 10 に示す String オブジェクトの詳細情報を確認すると，その String オブジェクトはファイルに保存されている盤面の状態を表している文字列であることが分かる．また，図 11 に示す Cell オブジェクトから CellMemento オブジェクトへの BASE の詳細情報を確認すると，その CellMemento オブジェクトのコンストラクタ実行中に Cell オブジェクトの getAlive メソッドを呼び出していることが分かる．同様に，図 12 に示す CellMemento オブジェクトから LinkedList オブジェクトへの BASE の詳細情報を確認すると，CellMemento オブジェクトを引数として LinkedList オブジェクトの offer メソッドを呼び出していることが分かり，図 13 に示す LinkedList オブジェクトから CellCaretaker オブジェクトへの BASE の詳細情報を確認すると，LinkedList オブジェクトが CellCaretaker オブジェクトのフィー

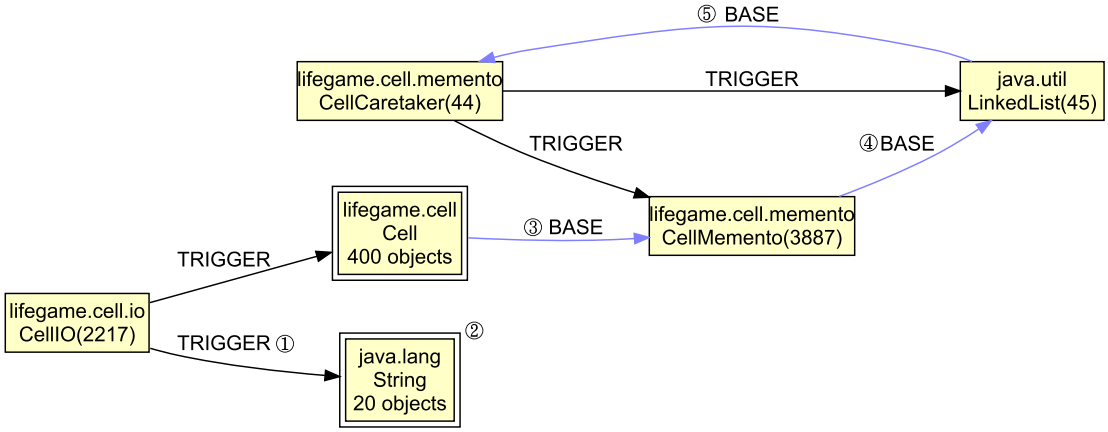

図 8: L4 の実行履歴から取得したオブジェクト生成関係 (一部抜粋)

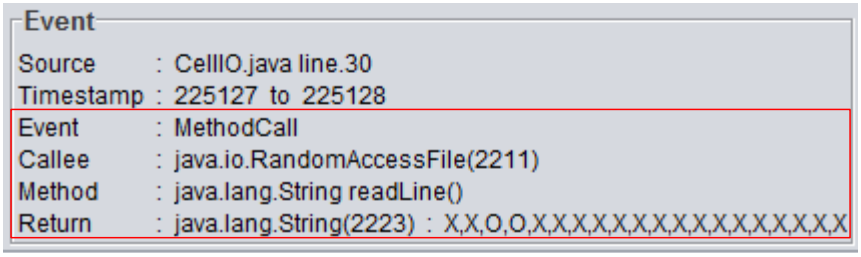

図 9: 図 8 の①の詳細ダイアログ (一部抜粋)

ルドに書き込まれていることが分かる．

図 8 のみから読み取ることはできないが，他の実行シナリオの実行履歴から取得したオブジェクト生成関係を見てみると，CellMemento オブジェクトは盤面の状態の履歴を表し，CellCaretaker オブジェクトが LinkedList オブジェクトを利用して，履歴である CellMemento オブジェクトを管理していることが分かる．

これらのことから，盤面の状態を保存しているファイルを開いた際には CellIO オブジェクトがファイルから読み込んだ盤面の状態を表す Cell オブジェクトを生成する．そして，その Cell オブジェクトからセルの状態を取得することで履歴を表す CellMemento オブジェクトが生成され，履歴を管理している CellCaretaker オブジェクトが持つ LinkedList オブジェクトに追加されていることが分かる．

実際の動作では盤面の状態を保存しているファイルを開くとファイルから読み込んだ状態の盤面が表示され，巻き戻し機能を利用した様子から読み込んだ盤面の状態は履歴に残っていないことが分かっている．したがって，盤面の状態を保存しているファイルを開いた際にはファイルから読み込んだ盤面の状態を一度履歴として記録し，巻き戻し機能を用いて表示

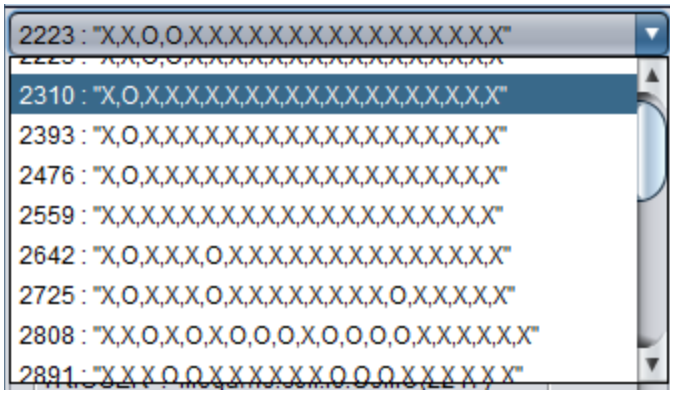

図 10: 図 8 の②の詳細ダイアログ (一部抜粋)

していると判断することができる．

この情報をソースコードのみから読み取る場合，セルの状態の読み込みがどこで実行されるかが分からないため，読み込み処理の開始から見ていくことになる．この場合，継承関係を考慮しつつ，25 クラスの該当箇所を読まなくてはならない．一方で本ツールを用いると，提示内容から上記の内容を理解し，実際に巻き戻し機能を利用しているかを確認するために 4 クラスの該当箇所を読むだけですんだ．したがって，提案手法を用いることでソースコードを読む量を減らすことができ，プログラム理解に役立てることができる．

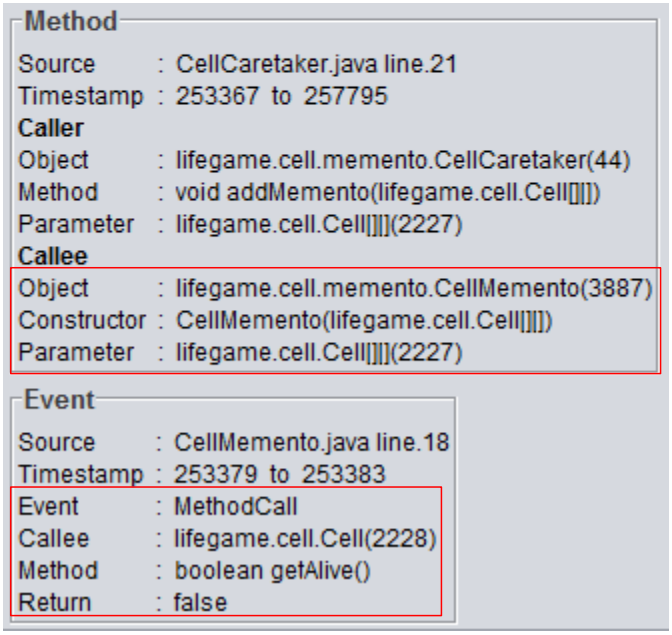

図 11: 図 8 の③の詳細ダイアログ (一部抜粋)

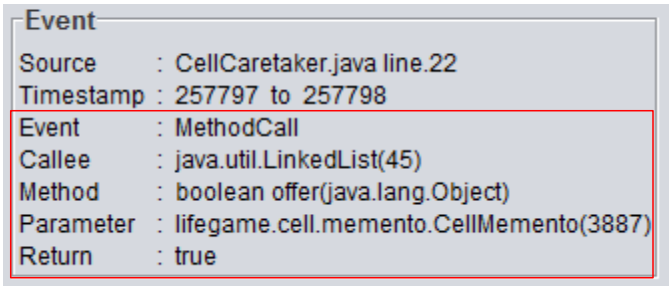

図 12: 図 8 の④の詳細ダイアログ (一部抜粋)

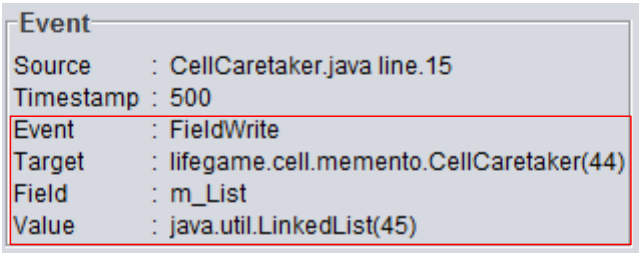

図 13: 図 8 の⑤の詳細ダイアログ (一部抜粋)

5.2 メソッドのテスト用データの作成

ライフゲーム L1 と L2 に対してメソッドのテストに必要なオブジェクトの生成を行った．具体的には各メソッドテストに必要なレシーバオブジェクトと引数のオブジェクトの生成方法を調べ，実際にオブジェクトを生成してメソッドのテストを行い，正しい結果が得られるかを確認した．

本実験の手順は以下の通りである．

1. 対象メソッドのソースコードを読み，どのような状態のオブジェクトが必要かを理解する．
2. 本手法を用いて対象メソッドのレシーバオブジェクトや引数のオブジェクトがどのように生成されたかを調査する．
   (a) プログラムを実行することで実行履歴を取得する．
   (b) 実行履歴の最初から対象メソッドが実行される直前までを指定区間としてオブジェクト生成関係を特定する．
   (c) 特定したオブジェクト生成関係からレシーバオブジェクトや引数のオブジェクトの生成に関係する部分グラフを抽出する．
   (d) 提示内容から生成方法を理解する．
3. 対象メソッドのテストを実行するコードを生成する．
4. 生成したコードを実行し，正しい結果が得られるかを確認する．

対象とするメソッドは表 4 に示すクラスに含まれるメソッドとした．L1 と L2 はガイドラインに沿って作成されているため，どちらのプログラムでも同名のクラスが存在している．

5.2.1 結果

本実験の結果を表 5 にまとめた．

表 4: 対象クラス

クラス	内容
BoardHistory	履歴を管理するクラス
BoardModel	盤面の状態を表すクラス
GameController	ゲーム全体を管理するクラス

著者は対象とした 61 メソッド中 46 メソッドのテストに必要なオブジェクトを生成することができた．生成できなかったもののほとんどはプリミティブ型のフィールド値や配列の値の変更方法が分からなかったことが原因である．たとえば，L1，L2 ともに BoardModel については半分以上のメソッドでテストに必要なオブジェクトを生成することができなかった．これは，BoardModel が各セルの状態を boolean 型の二次元配列で管理しているため，セルの状態を変化させる方法を本手法の提示内容から理解することができなかったからである．

5.2.2 考察

本手法ではオブジェクトがもつプリミティブ型の値の変更方法を理解することは困難である．なぜならば，本手法はオブジェクト生成におけるオブジェクト間の影響を特定するものであり，プリミティブ型のフィールドや配列に値が代入されたとしてもその内容はグラフ上には表示されないからである．

しかし，プリミティブ型の値の変更方法はソースコードから比較的容易に理解できると考えている．実際に今回プリミティブ型の値の変更方法を理解できなかったものについては，値を変更するためのメソッドが用意されており，ソースコードから比較的容易に理解することができた．したがって，本手法はオブジェクトの生成方法を理解する上で有効であると考えられる．

5.3 実行性能の調査

本ツールの実行性能を確かめるため，複数の実行履歴に対して解析を行った．

解析対象のプログラムは 6 つのライフゲームの他に Ant[2]，ANTLR[3]，DaCapo[6] を用いた．実行シナリオとしては Ant，および ANTLR は既存の JUnit[10] のテストケースを実行した．DaCapo は各ベンチマークを実行し，ライフゲームは実装されている機能を一通り

表 5: メソッドのテスト用オブジェクトの生成結果

プログラム	クラス	メソッド数	平均 LOC	生成できたメソッド数	割合 (%)
L1	BoardHistory	7	3.71	7	100.00
L1	BoardModel	7	8.57	3	42.86
L1	GameController	12	5.50	12	100.00
L2	BoardHistory	8	3.75	7	87.50
L2	BoardModel	10	9.50	4	40.00
L2	GameController	17	26.88	13	76.47

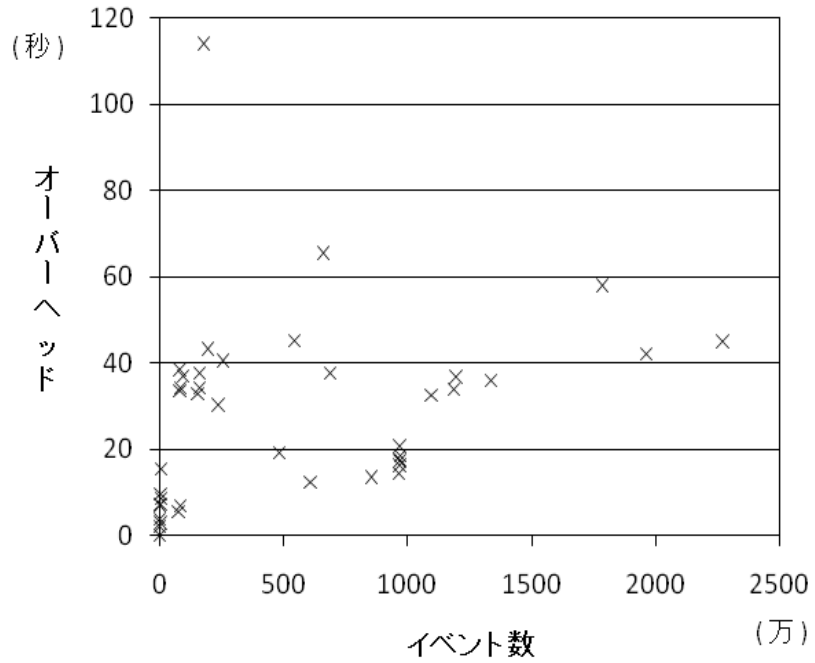

図 14: イベント数と実行履歴の記録によるオーバーヘッドの関係

実行した．ただし，実行履歴の記録の際に正常に動作しなかったものや記録したイベント数が膨大なため，実行履歴の読み込みの段階でメモリ不足になるおそれがあるものは解析の対象から除外した．

各実行履歴に対して，実行履歴全体のオブジェクト生成関係を特定し，集約を行うことで，生成関係の特定にかかった時間，集約にかかった時間，集約前後のグラフのサイズを調査した．また，ライフゲーム以外は実行履歴を記録する場合と記録しない場合の実行時間を調査した．ライフゲームは GUI プログラムであり，人による操作やタイマーによって実行される処理を再現することが困難であり，実行履歴の記録の有無による実行時間の比較を正確に行うことができないため，実行時間は調べていない．

5.3.1 結果

48 の実行履歴に対して調査を行った．

イベント数と実行履歴の記録によるオーバーヘッドの関係を図 14，イベント数と生成関係特定時間の関係を図 15，オブジェクト数と生成関係特定時間の関係を図 16，頂点数と集約時間の関係を図 18，頂点数と集約による頂点の減少数の関係を図 19，頂点数と集約後の頂点数の関係を図 20 に示す．

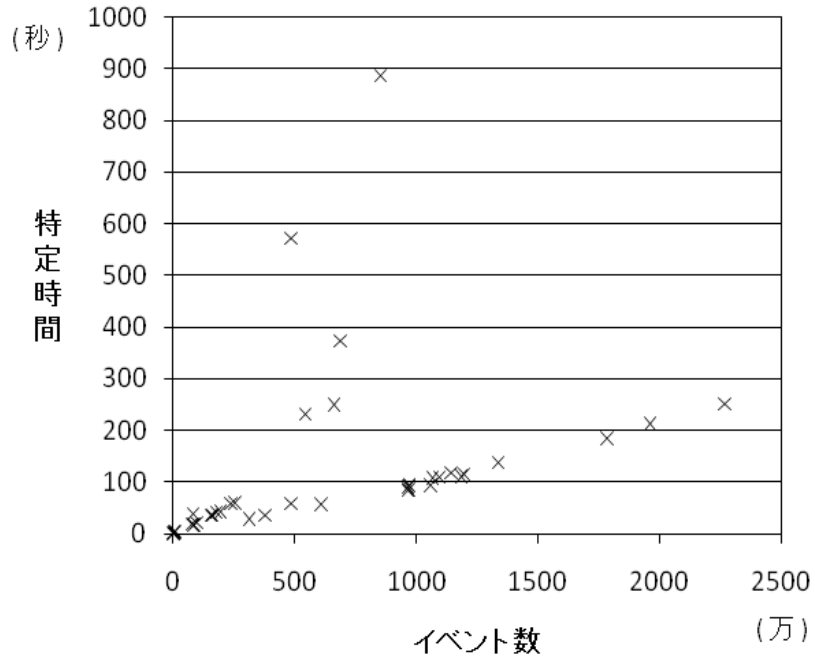

図 15: イベント数と生成関係特定時間の関係

5.3.2 考察

図 14 から分かるように，ほぼ同数のイベントを記録する場合でもオーバーヘッドは異なる．本手法の実行履歴はメソッドの引数の値などをすべて記録しているため，イベント数が同じでも記録するデータ量が異なるため，差が生じているのではないかと考えられる．また，通常の実行は平均 1.28 秒であることから，かなりのオーバーヘッドが生じていることが分かる．

図 15 と図 16 から分かるようにイベント数やオブジェクト数が増加すると特定時間も増加している．しかし，イベント数よりもオブジェクト数のほうが特定時間への影響が大きいように見える．相関係数を調べてみるとイベント数と特定時間の相関係数が 0.41，オブジェクト数と特定時間の相関係数が 0.64 であることからも，イベント数よりオブジェクト数のほうが特定時間への影響が大きいことが分かる．

本手法では区間を指定することでオブジェクト生成関係を特定するオブジェクト数や解析対象のイベント数を減らすことができるため，区間を指定することで短時間でオブジェクト生成関係を特定することができると考えている．本手法を利用する際に一度に特定するオブジェクト数は多い場合でも，数万程度を想定している．そのため，図 16 におけるオブジェクト数が 5 万以下の部分を拡大したものを図 17 に示す．このグラフをみると，オブジェク

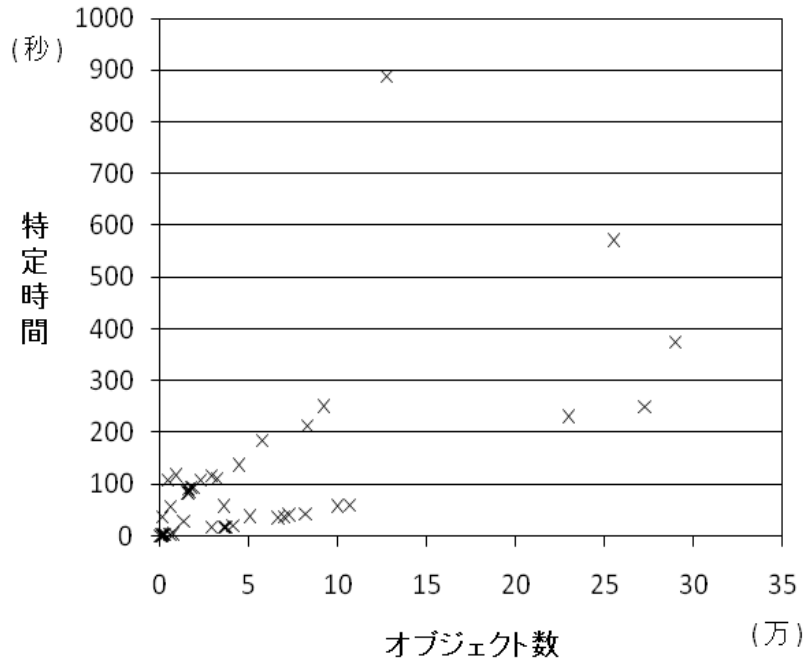

図 16: オブジェクト数と生成関係特定時間の関係

ト数が 5 万以下の場合はほとんどが 2 分以下，半分以上が 1 分以下でオブジェクト生成関係を特定できている．したがって，利用すると想定しているオブジェクト数では現実的な時間でオブジェクト生成関係を特定できると考えている．

次に図 18 より集約にかかった時間をみると一部で大きく時間がかかっているものが存在する．これは集約に時間がかかるグラフ形状が含まれているためだと考えられる．図 18 のオブジェクト数が 10 万以下の部分をみると短時間で集約が完了している．本手法はオブジェクト数が数万程度のものを扱うと想定しているため，現実的な時間で集約が完了すると考えられる．また，図 19 から頂点数が多いものでも集約によりほぼ同じ割合で頂点数が減っていることが分かる．図 20 から実際にほとんどの場合で大きく頂点数が減っていることが分かる．これはプログラムの実行時に同様の処理が繰り返し行われているためだと考えられる．そのため，集約は本手法で提示されるグラフのサイズを抑えるのに有効な手段と考えられる．

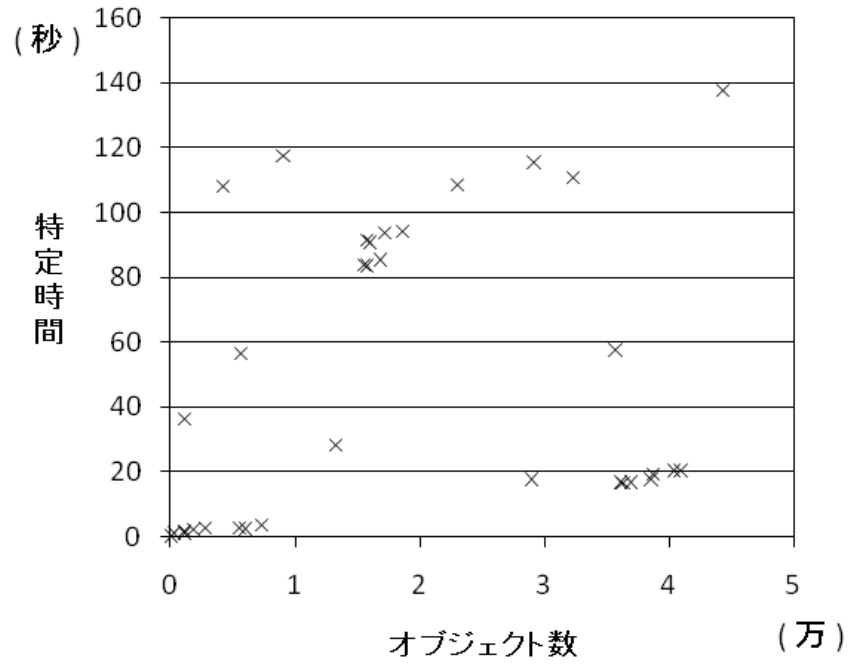

図 17: オブジェクト数と生成関係特定時間の関係 (拡大版)

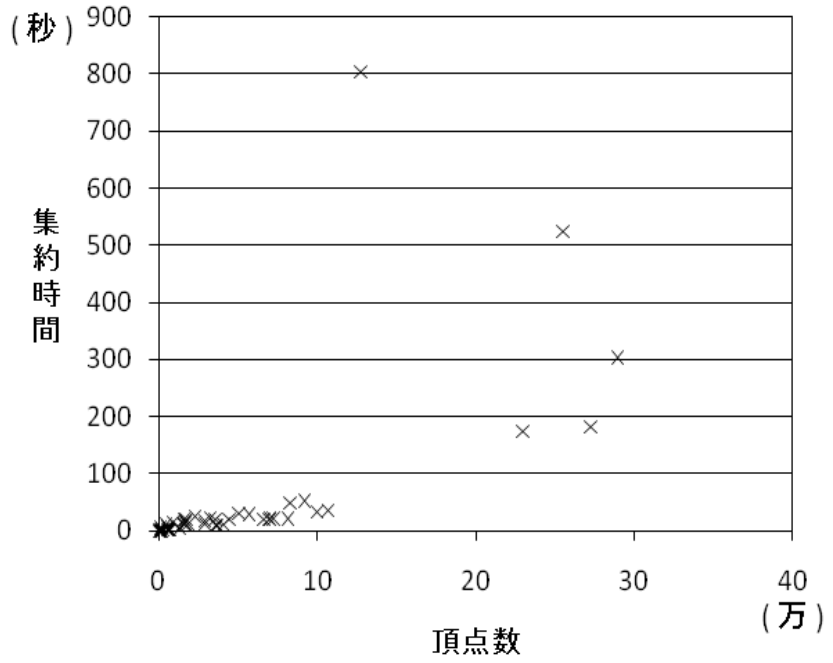

図 18: 頂点数と集約時間の関係

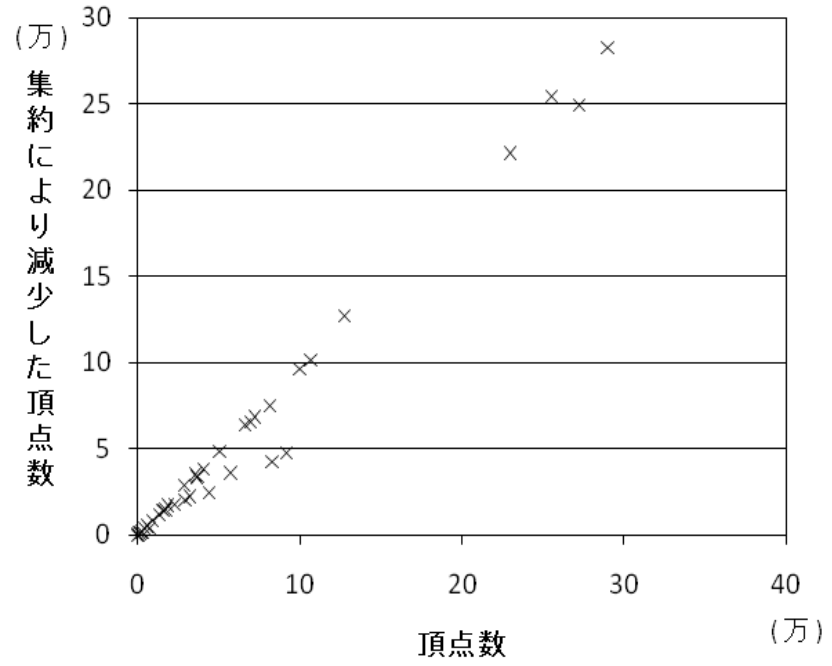

図 19: 頂点数と集約による頂点の減少数の関係

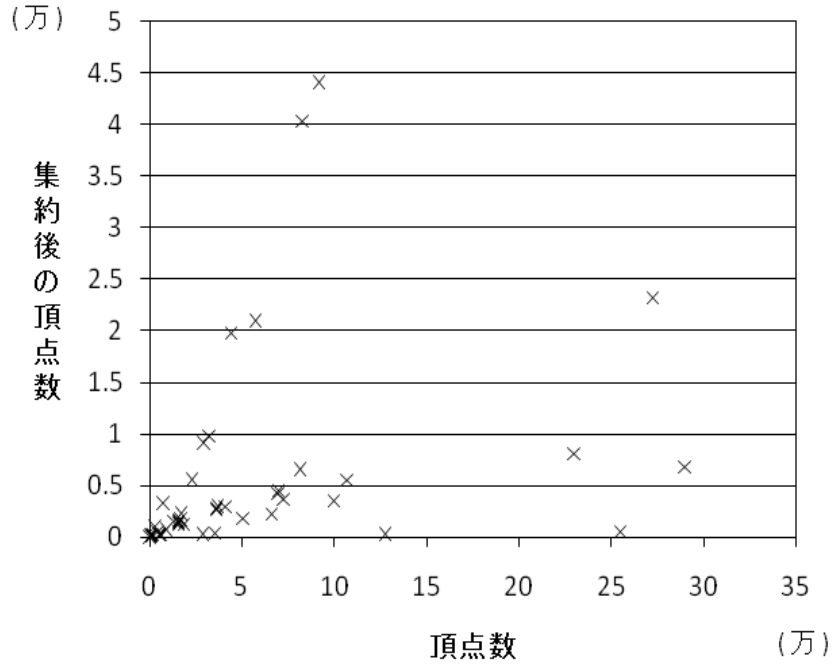

図 20: 頂点数と集約後の頂点数の関係

6　関連研究

オブジェクト間の関係を可視化する手法は複数提案されている．

オブジェクト間のメッセージのやり取りを UML のシーケンス図として可視化する手法が提案されている [25]．この手法を用いることでプログラムがどのように動作しているかを知ることができるが，オブジェクト間のメッセージのやり取りを時系列で示すことに主眼を置いているため，オブジェクト生成の様子を読み取ることはできない．

Rayside らは動的解析を利用し，実行中にどのオブジェクトがどのオブジェクトを保持していたかをオブジェクトの所有関係として特定し，可視化している [20]．この手法を用いることでオブジェクトがどのクラスのオブジェクトを持っていたかを判断できるが，本研究と違い，どのように生成されたかは判断することができない．

Lienhard らはオブジェクトがどのオブジェクトを経由して移動し，どのオブジェクトで保持されているかを可視化している [14]．この手法を用いることにより，オブジェクト生成時にどのオブジェクトにアクセスできたかを知ることができるが，どのように利用されたかまでは知ることができない．

Lienhard らはメソッドの実行中に利用する参照関係とそのメソッド実行によって変化する参照関係を可視化する手法も提案している [15]．この手法を用いることでメソッド実行に必要なオブジェクトの状態やメソッド実行による状態変化を知ることができるが，メソッドの実行に必要な状態のオブジェクトを生成する方法は分からず，そのため，メソッドによる状態変化を知ってもオブジェクトの生成に生かすことができない．

また，再利用を目的として利用者が理解しているクラスと利用したいクラスをクエリとして指定することで理解しているクラスのオブジェクトから利用したいクラスのオブジェクトを取得する方法を提示する手法が提案されている．Mandelin らは API を用いて，あるオブジェクトからメソッド呼び出しやフィールドアクセスなどを行うことで取得可能なオブジェクトを特定することで，利用者が指定したクエリを満たすメソッド列の候補を提示する手法を提案している [17]．Thummalapenta らはコードサーチエンジンを用いることで利用者が指定したクエリを満たすコードサンプルを集め，コードサンプルから余分な部分を除去したメソッド列を候補として提示する手法を提案している [23]．これらの手法は静的に行われているため，候補となるメソッド列を提示できない場合があり，メソッド列を提示できたとしても確実に利用したいクラスのオブジェクトを取得できるわけではない．また，取得するオブジェクトの状態は考慮していない．本研究は動的解析を用いており，静的解析よりも正確に生成方法を調べることが期待でき，利用したいクラスのある特定状態オブジェクトに絞って調査することができる．

7　まとめ

本研究ではオブジェクトがどのように生成されるかを知ることでプログラム理解作業を支援できると考えた．しかし，オブジェクトの生成に他のオブジェクトを利用する場合，利用するオブジェクトも同様に生成方法を調べる必要があるため，オブジェクトの生成方法を理解することは困難である．そこで，オブジェクトがどのように生成されたかをオブジェクト生成関係として可視化する手法を提案した．本手法を用いることである特定状態のオブジェクトを生成するために，どのオブジェクトがどのような影響を与えたかを知ることができる．Java プログラムを対象として，提案手法をツール化し，ケーススタディを実施した．ケーススタディでは，シナリオの分析を行うことでプログラムの動作理解を行う際の実例を示し，メソッドのテスト用オブジェクトを生成することでオブジェクトの生成方法を理解できることを示した．また，オブジェクト生成関係の特定時間と集約の効果を調査し，利用を想定しているオブジェクト数では現実的な時間でオブジェクト生成関係を特定することができ，集約は本手法で提示するグラフのサイズを抑えるために有効であることを示した．

今後の課題を次に挙げる．

- 大規模プログラムへの適用
  本研究では小規模なプログラムに対してしか適用しておらず，大規模プログラムに対しても適用する必要がある．

- 提案手法の客観的評価
  本研究では著者しか提案手法を利用していないため，被験者を用いた評価を行う必要がある．

- 区間指定の支援
  本研究ではメソッド実行の区間を提示することで支援を行ったが，利用者が注目したい部分がどのメソッドで行われているかを知らなければ，利用することはできない．そのため，利用者が注目したい区間を指定するための支援が必要である．

- 提示するグラフのサイズ対策
  提示するグラフのサイズが大きくなると利用者が読解することは困難になる．本手法では集約や特定のオブジェクトの生成に関係する部分グラフの抽出などで対策を行っているが，それでも提示するグラフが巨大になる場合がある．そのため，利用者が注目したい部分を残しつつ，グラフのサイズを抑えるためのさらなる方法が必要である．

謝辞

本研究において，常に適切な御指導および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いたします.

本研究において，適切な御指導および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 准教授に深く感謝いたします.

本研究において，逐次適切な御指導および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　石尾 隆 助教に深く感謝いたします.

本研究において，様々な御指導および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　渡邊 結 氏に深く感謝いたします.

本研究において，様々な御指導および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　伊達 浩典 氏に深く感謝いたします.

最後に，その他様々な御指導および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] Amida. http://sel.ist.osaka-u.ac.jp/~ishio/amida/.

[2] Ant. http://ant.apache.org/.

[3] ANTLR. http://www.antlr.org/.

[4] G. Arevalo, F. Buchli, and O. Nierstrasz. Detecting implicit collaboration patterns. In *Proceedings of the 11th Working Conference on Reverse Engineering*, pp. 122–131, November 2004.

[5] A. S. Boujarwah, K. Saleh, and J. Al-Dallal. Dynamic data flow analysis for java programs. *Information and Software Technology*, Vol. 42, pp. 765–775, August 2000.

[6] DaCapo. http://www.dacapobench.org/.

[7] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *Proceedings of the 2006 International Workshop on Dynamic Analysis*, pp. 17–23, May 2006.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley, 1995.

[9] Graphviz. http://www.graphviz.org/.

[10] JUnit. http://www.junit.org/.

[11] T. D. LaToza and B. A. Myers. Developers ask reachability questions. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pp. 185–194, May 2010.

[12] M. Lejter, S. Meyers, and S. P. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions on Software Engineering*, Vol. 18, No. 12, pp. 1045–1052, December 1992.

[13] H. Lieberman and C. Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, Vol. 26, No. 6, pp. 419–429, June 1983.

[14] A. Lienhard, S. Ducasse, and T. Girba. Taking an object-centric view on dynamic information with object flow analysis. *Computer Languages, Systems & Structures*, Vol. 35, pp. 63–79, April 2009.

[15] A. Lienhard, T. Girba, O. Greevy, and O. Nierstrasz. Test blueprints – exposing side effects in execution traces to support writing unit tests. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering*, pp. 83–92, April 2008.

[16] A. Lienhard, O. Greevy, and O. Nierstrasz. Tracking objects to detect feature dependencies. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pp. 59–68, June 2007.

[17] D. Mandelin, L. Xu, R. Bodik, and D. Kimelman. Jungloid mining: Helping to navigate the api jungle. In *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pp. 48–61, June 2005.

[18] M. Pradel and T. R. Gross. Automatic generation of object usage specifications from large method traces. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, pp. 371–382, November 2009.

[19] J. Quante and R. Koschke. Dynamic object process graphs. *Journal of Systems and Software*, Vol. 81, pp. 481–501, April 2008.

[20] D. Rayside, L. Mendel, and D. Jackson. A dynamic analysis for revealing object ownership and sharing. In *Proceedings of the 2006 International Workshop on Dynamic Analysis*, pp. 17–23, May 2006.

[21] T. Richner and S. Ducasse. Using dynamic information for the iterative recovery of collaborations and roles. In *Proceedings of the 18th IEEE International Conference on Software Maintenance*, pp. 34–43, October 2002.

[22] T. Systa. Understanding the behavior of java programs. In *Proceedings of the 7th Working Conference on Reverse Engineering*, pp. 214–223, November 2000.

[23] S. Thummalapenta and T. Xie. Parseweb: A programmer assistant for reusing open source code on the web. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pp. 204–213, November 2007.

[24] G. Xu, M. Arnold, N. Mitchell, A. Rountev, and G. Sevitsky. Go with the flow: Profiling copies to find runtime bloat. In *Proceedings of the ACM SIGPLAN 2009 Conference on Programming Language Design and Implementation*, pp. 419–430, June 2009.

[25] 谷口考治, 石尾隆, 神谷年洋, 楠本真二, 井上克郎. プログラムの実行履歴からの簡潔なシーケンス図の生成手法. コンピュータソフトウェア, Vol. 24, No. 3, pp. 153–169, 2007.

付録

A ライフゲーム

ライフゲームとはコンウェイによって考案されたシュミレーションゲームである．

碁盤目状の格子を盤面とし，各格子「セル」は「生きている」か,「死んでいる」状態のどちらかをとる．なお，盤面の外側はすべて死んでいるセルとして扱う．このゲームでは盤面の全セルに対して，以下のルールを適用することを「1 世代進める」と表現する．

セルの生存条件 生きているセルの周囲 8 セルに 2 つ，または 3 つの生きているセルがある
ある生きているセルは，その周囲に 2 つ，または 3 つの生きているセルがあるとき，次の世代でも生きている状態が続く．周囲に生きているセルが 1 つ以下の場合は孤立により，4 つ以上の場合は過密により，そのセルは次の世代で死んでいる状態になる．

セルの誕生条件 死んでいるセルの周囲 8 セルに 3 つの生きているセルがある
ある死んでいるセルの周囲に 3 つの生きてるセルがあるとき，次の世代で生きている状態になる．この条件を満たさない死んでいるセルは，次の世代でも死んだ状態のままである．

A.1 基本仕様

5 章では L1 から L6 までの 6 つのライフゲームプログラムを用いた．これらのライフゲームプログラムは大阪大学基礎工学部情報科学科の 2 年生が演習で作成したプログラムである．L4 のスクリーンショットを図 21，図 21 から 1 世代進めたものを図 22 に示す．このライフゲームの基本仕様は以下の通りである．

GUI による盤面の表示

プログラムを起動すると，全セルが死んでいる状態の盤面をウィンドウ上に表示する．盤面は，正方形のセルを垂直方向と水平方向に同数並べた正方形の格子として表現する．ウィンドウは「Next」,「Back」,「Auto」の 3 つのボタンを持つ．

マウスによる盤面の編集

マウスの左ボタンの押下，および押したままでのマウスカーソル移動により，盤面上のセルの生死状態を切り替える．あるセルの生死状態の切り替わりが発生するのは，そのセル上でマウスの左ボタンを押下したときと，左ボタンが押された状態でマウスカーソルが隣接セルから侵入したときだけである．ただし，盤面の外側でマウスの左ボタンが押され，そのままの状態でセルに侵入した場合は生死状態の切り替えは行わない．

「Next」押下による世代の更新

「Next」ボタンを押すごとに，盤面の状態を 1 世代進める．

「Back」押下による盤面の巻き戻し

「Back」ボタンを押すごとに，盤面の状態を 1 つ巻き戻す．世代が 1 つ進む，またはマウスにより 1 つのセルの生死状態が切り替わるごとに盤面の状態が変化するとして，現在の状態から 16 前の状態までを履歴として記憶しておく．また,「Back」ボタンは巻き戻しが可能な状態でのみ有効にする．

「Auto」による自動更新

「Auto」ボタンを押すと，200 ミリ秒につき 1 世代程度の速度で，自動的に盤面を更新していく．自動更新中は「Auto」ボタンを「Stop」ボタンに変更し,「Stop」ボタンを押すと自動更新を停止し,「Auto」ボタンへ戻す．自動更新中も，他のボタン操作やウィンドウの移動などの操作も受け付ける．

A.2 拡張仕様

各ライフゲームプログラムは課題としてそれぞれ基本仕様以外の機能を実装している．ここでは 5.1 節で利用した L4 と 5.2 節で利用した L1，L2 における拡張仕様について説明する．

セルの状態数の表示 (L1，L2，L4)

現在の盤面における以下のセルの数をウィンドウ上に表示する．

- 生きているセル (L1，L2，L4)
- 1 つ前の状態で生きており，現在も生きているセル (L1，L4)
- 1 つ前の状態で死んでおり，現在は生きているセル (L1，L2，L4)
- 1 つ前の状態で生きており，現在は死んでいるセル (L1，L2，L4)

盤面のクリア (L1，L2)

盤面上の全セルを死んでいる状態に変更し，履歴を削除する．

無変化の世代更新の禁止 (L2)

現在の状態が次の世代の状態と同じ場合は世代の更新を受け付けずに，その旨を伝えるダイアログを表示する．自動更新中の場合は自動更新を停止し,「Stop」ボタンを「Auto」ボタンに戻す．ダイアログの表示，非表示は設定で自由に切り替えることができる．

既存の盤面の表示 (L2)

プログラム中に特定状態の盤面を用意しておき，この盤面を現在の状態とすることができる．この機能を利用した際には履歴を削除する．

設定の変更 (L2，L4)

以下の内容を自由に設定できる．

- 履歴の記憶数 (L2，L4)
- 盤面のセル数 (L2，L4)
- セルのサイズ (L4)
- ルールの誕生条件 (L4)
  死んでいるセルの周囲 8 セルに 3 つの生きているセルがある
  死んでいるセルの周囲 8 セルに 3 つ，または 6 つの生きているセルがある

設定の表示 (L4)

ウィンドウ上に現在の設定値を表示する．

設定の保存 (L4)

現在の設定をファイルに自動的に保存する．次回起動時はファイル記録された設定を利用する．

盤面の状態の保存 (L4)

現在の盤面の状態をファイルに保存できる．

ファイルに保存した盤面の状態を読み込み，現在の状態とすることができる．

メニューバーからの Next，Back の実行 (L2)

ボタンの代わりにメニューバーからも世代の更新や世代の巻き戻しを実行できるようにする．

ショートカットキー (L2)

盤面のクリア，盤面のセル数の変更，プログラム終了，世代の更新，世代の巻き戻しに対応するショートカットキーを用意する．

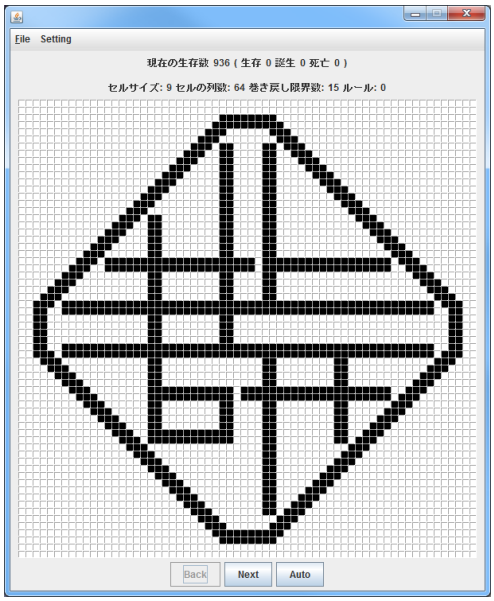

図 21: ライフゲーム (L4) のスクリーンショット

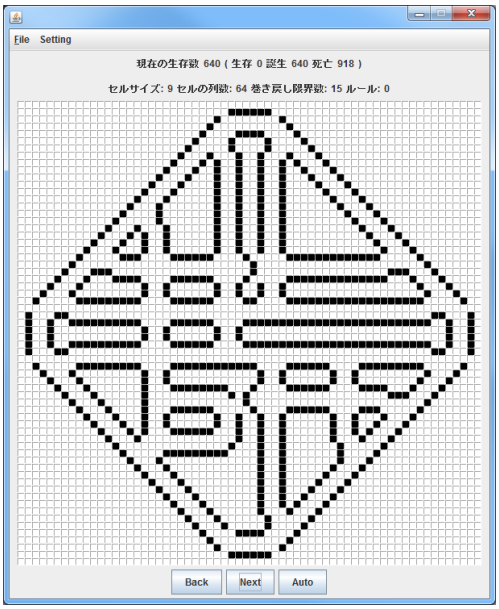

図 22: 図 21 から 1 世代進めたスクリーンショット

B 指定オブジェクトの生成に関係する部分グラフの抽出

指定オブジェクトの生成に関係する部分グラフの抽出を行う際に，本研究では 3.6 節で述べた基準を用いて BASE をたどった．しかし，この基準には不備があることが判明した．たとえば，オブジェクト A のフィールドにオブジェクト B を書き込み，書き込んだ後にオブジェクト C のデータを用いて生成したデータをオブジェクト B のフィールドに書き込んだとする．このとき，BASE は $B \xrightarrow{BASE-A} A$ と $C \xrightarrow{BASE-U} B$ の 2 つが特定される．$B \xrightarrow{BASE-A} A$ を特定した際のイベントのタイムスタンプを x，$C \xrightarrow{BASE-U} B$ を特定した際のイベントのタイムスタンプを y とすると，$x < y$ が成立する．オブジェクト A の生成に関係する部分を抽出するとき，オブジェクト B と C はオブジェクトの生成に必要なオブジェクトと判断すべきである．しかし，3.6 節で述べた基準を用いてオブジェクト A の生成に必要なオブジェクトを特定すると，オブジェクト A から $B \xrightarrow{BASE-A} A$ をたどった後，BASE-U であり，かつ $x < y$ であるために $C \xrightarrow{BASE-U} B$ をたどることはない．したがって，オブジェクト C はオブジェクト A の生成に必要ないと判断されてしまう．

そこで，BASE をたどる際の基準を以下のように修正すべきである．

- 現在までに BASE-A しかたどっていない場合
  - BASE-A，BASE-U ともに無条件にたどる
- 現在までに BASE-U を 1 回でもたどった場合
  - BASE-A，BASE-U ともに直前にたどった BASE のタイムスタンプよりも前のタイムスタンプを持つもののみをたどる

上記の基準を用いた際の例を図 23 に示す．元となるグラフは図 2 で用いたものと同じである．

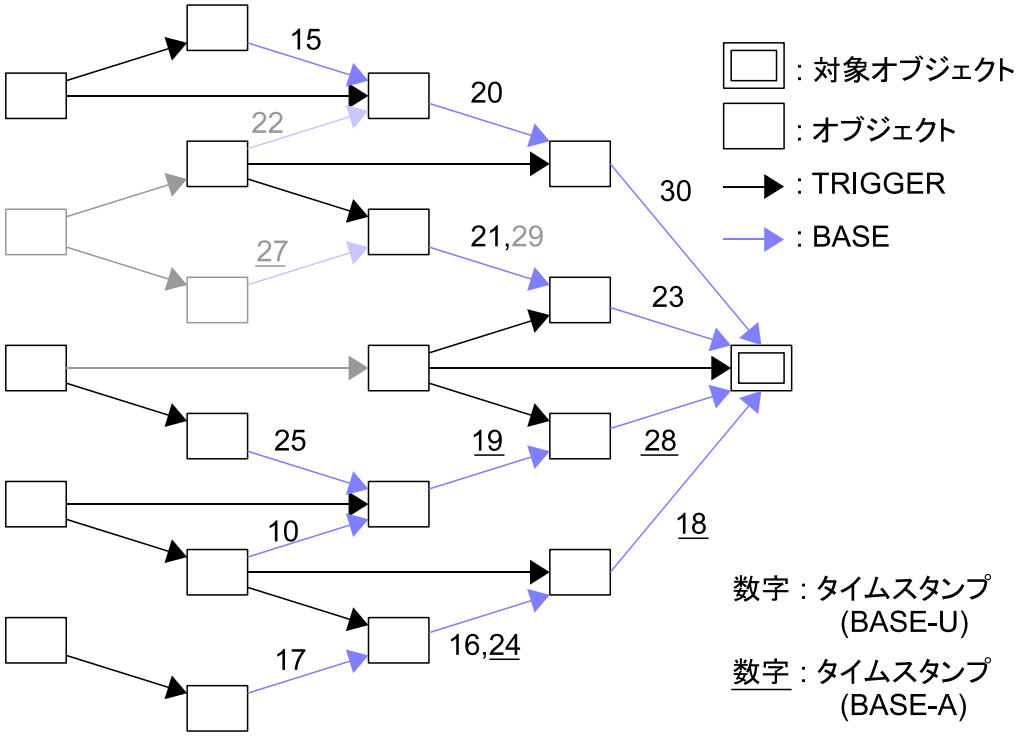

図 23: 修正した基準を用いた際の抽出例