情報処理学会論文誌 Vol.53 No.3 1166–1176 (Mar. 2012)

# 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

中野 佑紀[1] 伊達 浩典[1] 渡邊 結[1] 石尾 隆[1,a)] 井上 克郎[1]

受付日 2011年3月28日, 採録日 2011年12月16日

**概要**：オブジェクト指向プログラムは，1つの処理を実行するために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うえで有用である．本研究では，オブジェクト指向プログラムの実行履歴から，オブジェクト間の生成関係を抽出し，有向グラフとして可視化する手法を提案する．提案手法の有効性を評価するために，ケーススタディとして，大学の演習科目で作成されたJavaプログラムの単体テストに必要なオブジェクトの生成を行うコードを，可視化された生成関係に基づいて記述した．その結果，61個のメソッドのうち，46個のメソッドのテストに必要なオブジェクトを生成するコードを記述することができ，可視化されたグラフからオブジェクトの生成関係を読み取れることを確認した．

**キーワード**：プログラム理解，動的解析，ソフトウェア可視化

# Visualizing Relationship of Object Generation Based on Execution Trace

Yuki Nakano[1] Hironori Date[1] Yui Watanabe[1] Takashi Ishio[1,a)]
Katsuro Inoue[1]

Received: March 28, 2011, Accepted: December 16, 2011

***Abstract:*** Object-oriented programs generate many temporary objects to execute a functionality. Understanding how objects are generated is useful to understand the behavior of a program. In this research, we propose a technique to identify relationship of object generation; for each object, we visualize a set of objects which affects the object construction. To evaluate our approach, we have conducted a case study on Java programs written by undergraduate students. The visualized object generation relationship enabled us to write code to generate objects for test cases for 46 of 61 methods in the programs.

***Keywords:*** program comprehension, dynamic analysis, software visualization

## 1. まえがき

ソフトウェアのデバッグや保守作業では，その対象となっているソフトウェアの構造や振舞いを正しく理解することが重要である．LaTozaら [1] は，保守作業に要する時間の多くはプログラム理解に費やされていると述べており，また，開発者が頻繁に行う作業の1つとして，オブジェクトの生成に関する処理の調査をあげている．オブジェクト指向プログラムにおいて，重要なデータはすべてオブジェクトとして管理されているため，オブジェクトの内容を調べることが，プログラム理解において重要であると考えられる．

オブジェクト指向プログラムの実行中に，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．なぜなら，オブジェクトの生成とは，オブジェクト用のメモリ領域を初期化するだけでなく，そのあとにオブジェクトに適切なデータを格納する一連の処理が続く場合があるためである．たとえば，GUIアプリケーションにおけるウィンドウの生成方法を知りたい場合，コンストラクタを呼び出すことで空のウィンドウを作成するという処理だけでなく，そこに適切な部品を追加す

[1] 大阪大学大学院情報科学研究科
Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[a)] ishio@ist.osaka-u.ac.jp

る処理の完了までを調査する必要がある．また，動的束縛を使用したオブジェクト指向プログラムの実行の流れは局所的なソースコードの情報のみでは確定しないことがあるため，単純にソースコードを追跡するだけでは，生成処理の理解は困難である．

そこで本研究では，オブジェクト指向プログラムの実行履歴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，あるオブジェクトについて，そのオブジェクトのコンストラクタを呼び出したオブジェクトや，オブジェクトに格納したデータを提供したオブジェクトを特定し，オブジェクトを頂点，制御とデータの流れを辺としたオブジェクト生成関係グラフを抽出する．

本研究で提案する生成関係の有効性を評価するために，大学の演習科目で作成されたプログラムに対して提案手法を適用し，プログラムの単体テストに必要なオブジェクトを生成するコードを記述するケーススタディを実施した．第1著者が，テスト対象となっているクラスおよびテストが完了したクラス以外のコードの閲覧を禁止した状態で，可視化された生成関係を閲覧しながらテストの作成作業を行ったところ，61個のメソッドのうち，46個のメソッドのテストに必要なオブジェクトの生成手順を理解することが可能であった．この結果は生成関係という情報の有用性を示している．

本論文の貢献は，次の2点である．

- オブジェクトの生成関係という概念を定義し，Javaプログラムの実行履歴から生成関係を自動抽出する動的解析手法を提案した．
- 提案手法の定義では，ライブラリなど，実行履歴を取得できない対象の取扱いを明示した．これは，Javaプログラムの動的解析を試みる研究者にとって有用な情報である．

本論文の構成は次のとおりである．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章では実施したケーススタディについて説明する．5章では関連研究を紹介する．最後に6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

プログラムのデバッグや保守作業において，開発者が知りたい情報とは，振舞いを調査しようとしている処理の実行に必要なオブジェクトの状態である．デバッグ作業では，ある時点で不正な状態のオブジェクトを発見した場合に，そのオブジェクトの生成に関与したオブジェクトを特定し，その不正な状態の原因を調査する必要がある．また，保守作業においても，処理に必要なデータを持つオブジェクトがいつ，どのように生成されているかを調べる必要がある．

本研究において，オブジェクト生成とは，オブジェクトのためのメモリ領域を確保し，適切なデータを格納することで，外部から利用可能な特定の状態を持ったオブジェクトを生成することと定義する．オブジェクトは，生成後も状態が変化することはあるが，最初に外部からデータの参照が行われるまでを，オブジェクトの生成と考える．そして，オブジェクトの生成を調査するという作業は，その生成処理を実行したオブジェクトと，その過程で参照されたオブジェクト群を特定する作業となる．たとえば，GUIアプリケーションにおけるウィンドウの生成に関係したオブジェクトとは，ウィンドウのコンストラクタを呼び出したオブジェクトや，ウィンドウに追加されたGUI部品オブジェクトのことを指す．

オブジェクトの生成を調査する作業は，プログラムのある時点から，過去にデータフローをさかのぼっていく作業となる．このようなデータフローを調査する手法の1つとして，プログラムスライシングが知られている [11]．しかし，プログラムスライシングは，次のような理由から，オブジェクトの生成を調査する作業に適していない．

- オブジェクトの生成は，コンストラクタの実行だけでなく，それ以降にデータ登録などの処理が実行されることが多い．オブジェクトへの参照を格納した変数に関するデータフローだけをたどっていくと，その参照変数の値を決定したコード位置（Javaの場合は `new` の結果を代入した位置）を追跡することは容易であるが，単純に `new` の位置だけを特定しても，生成が完了していない状態のオブジェクトしか得られない．
- オブジェクトは，生成完了後，外部から様々なデータ要求を受け取り，外部にデータを提供するため，オブジェクト間のデータ依存関係が複雑になる．オブジェクトに格納されたデータの由来を直接調査しようとしても，生成関係以外のデータフロー情報が混在した状況で調査を行わなければならない．

本研究では，実行時情報を用いて，各オブジェクトに対して，そのオブジェクトの生成に関わったオブジェクトを特定する．プログラムスライシングは，開発者が興味ある特定のデータに関するすべての命令を取り出す手法となっているが，本研究では，命令の列ではなく，特定のオブジェクトの生成に関与したオブジェクトの集合を提示することで，開発者に対して，オブジェクト生成に関する情報を簡潔に提示することを目指す．動的解析を用いた理由は，同じクラスのオブジェクトであっても，その状態の違いにより，プログラム内部でのオブジェクトの使用方法が異なる場合があるためである [7]．そのようなオブジェクトの区別を，静的解析では正確に行うことができない [5] ため，本研究では，実行時情報を使って，生成されたオブジェクトをそれぞれ区別して取り扱う．

## 3. 提案手法

本研究では，開発者が理解したいプログラムの実行を観測し，オブジェクトの生成関係を有向グラフとして可視化する手法を提案する．対象言語には Java を選択した．

2 つのオブジェクト $A$，$B$ について，オブジェクト $A$ がオブジェクト $B$ の生成に関与する方法を，次の 3 つの有向辺で表現する．

- $A \xrightarrow{TRIGGER} B$：オブジェクト $A$ が，コンストラクタ呼び出しか，それに類するメソッドの呼び出しを実行し，オブジェクト $B$ を構築した．
- $A \xrightarrow{BASE-A} B$：オブジェクト $A$ がオブジェクト $B$ のフィールドに格納された．
- $A \xrightarrow{BASE-U} B$：オブジェクト $A$ が，オブジェクト $B$ に何らかのデータを渡した．

TRIGGER 辺はオブジェクトの生成に関する制御フローを，BASE-A 辺と BASE-U 辺はオブジェクト間のデータフローを表現している．TRIGGER 辺の順序は時系列を反映しており，古いオブジェクトから，新しいオブジェクトへと辺が引かれる．BASE-A 辺と BASE-U 辺については，多くの場合は，辺が時系列を表現するが，厳密な順序関係は保証されない．たとえば，オブジェクト A がコンストラクタ内部で “`this.f = new B();`” という文を実行した場合は，$A \xrightarrow{TRIGGER} B$，$B \xrightarrow{BASE-A} A$ という循環依存が生じる．

このグラフを用いて，あるオブジェクトから矢印の向きを逆向きにたどっていくと，そのオブジェクトの生成に関与したすべてのオブジェクトを得ることができる．本研究では，実行履歴を用いてグラフを作成するため，各辺に対応する実行時のイベント情報を閲覧することで，開発者は，いつ，どのように，オブジェクトが生成されたかを知ることができる．

以降の説明では，$A \xrightarrow{TRIGGER} B$ となる A，B について「A は B の TRIGGER である」，$A \xrightarrow{BASE-A} B$ あるいは $A \xrightarrow{BASE-U} B$ であれば「A は B の BASE である」という表現を用いる．

### 3.1 実行履歴のモデル

本研究では，Java プログラムの実行履歴を，メソッド呼び出しと，フィールドおよび配列へのアクセスの列であると考えて解析を行う．観測するイベントは，以下のとおりである．

- メソッド，コンストラクタの呼び出しの前後で，現在処理を実行中（呼び出し側）のオブジェクトとメソッド，呼び出しの引数となったオブジェクト，戻り値（あるいは発生した例外）オブジェクトを記録する．
- メソッド，コンストラクタの実行開始と終了で，処理を実行する（呼び出された側）オブジェクトとメソッドを記録する．
- フィールドおよび配列に対する読み書きの直前と直後に，処理を実行したオブジェクトとメソッド，読み書き対象のオブジェクト，読み書きした値を記録する．

すべての実行イベントは，スレッドに関係なく一意に順序を特定するタイムスタンプを付与しておく．コンストラクタ呼び出しは，戻り値が void 型のメソッド呼び出しとして表現する．また，クラスメソッド（static メソッド）の呼び出しやクラスフィールド（static フィールド）に対する読み書きは，それぞれのクラスに対してただ 1 つ存在するクラスオブジェクトを対象とした操作であると見なす．引数および戻り値に登場する文字列オブジェクトに関しては，生成関係を特定した後の分析を容易にするために，その値を記録している．また，値がプリミティブ型であった場合は，実行履歴のサイズを処理可能な範囲に抑えるという実装の制約上，型のみを記録している．

このイベント履歴から，オブジェクト生成関係の特定に必要な情報として，メソッド呼び出し *call*，フィールドおよび配列の読み書き *access* を次のように抽出する．

メソッド呼び出し *call* は，9 項組 $\langle t_{call}, t_{ret}, o_1, m_1, o_2, m_2, P, s, r\rangle$ である．$t_{call}$ は呼び出しが生じた時刻，$t_{ret}$ は呼び出しが終了した時刻である．$o_1$，$m_1$ はそれぞれ呼び出し側のオブジェクト，メソッドであり，$o_2$，$m_2$ は呼び出し先のオブジェクト，メソッドである．$P = \{p_1, \ldots, p_n\}$ は，呼び出しの引数に登場したオブジェクトのリストである．$s = \{return, exception\}$ は，呼び出しが正しく終了したとき $return$，例外が発生したとき $exception$ となる．$r$ は，メソッドの戻り値あるいは発生した例外のオブジェクトを格納し，戻り値がオブジェクト以外の場合は型名を保持する．これらの情報は，通常，メソッドの呼び出し側と，それに対応して発生したメソッド実行のイベントから抽出する．

本研究では，開発者が指定したライブラリのメソッドの実行内容は，実行履歴の取得対象外とする．ライブラリへのメソッド呼び出しと，ライブラリから解析対象部分へのメソッド呼び出しだけを記録しておき，ライブラリの動作と見なす．たとえば，オブジェクト $o_1$ のメソッド $m_1$ が，java.util.HashSet クラスのオブジェクト $o_2$ に対して `add` メソッドを呼び出し，引数としてオブジェクト $o_3$ を渡したとする．この `add` メソッドが，内部で $o_3$ に対して `hashCode` メソッドを呼び出したとすると，観測できるのは $o_1$ から $o_2$ への呼び出しイベントと，$o_3$ のメソッド実行イベントとなる．このとき，$o_1$ のメソッド $m_1$ が $o_2$ の `add` を，$o_2$ の `add` メソッドが $o_3$ の `hashCode` メソッドをそれぞれ呼び出したと解釈し，2 つの *call* を抽出する．このとき，$o_2$ が内部で他のオブジェクトやメソッドを使っていたとしても，その内容は無視する．この方式は，ライブラリに所属するオブジェクトを呼び出したとき，ライブラリ全

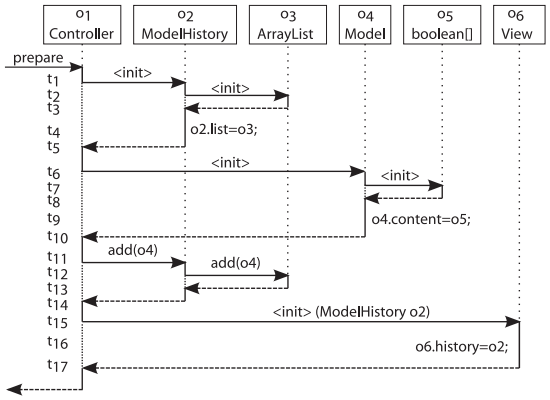


図 1 実行履歴の例
**Fig. 1** An example of execution trace.

体の動作をその 1 オブジェクトが担当していると見なして動作を近似するものとなっており，他の動的解析手法 [4] で用いられているものと同様である．

フィールドや配列のデータへのアクセスは，7 項組 $\langle t_{access}, op, d, o_1, m_1, o_2, v\rangle$ によって表現される．$t_{access}$ は，フィールドあるいは配列への読み書きの直前に対応するイベントの時刻である．$op = \{read, write\}$ は，読み書きの種類を表す．$d = \{field, array\}$ は，アクセスしたデータがフィールドの場合 $field$，配列の場合は $array$ という値をとる．$o_1$ はアクセスを行ったオブジェクト，$m_1$ はそのとき実行中だったメソッド，$o_2$ はアクセスされたオブジェクトをそれぞれ表現する．$v$ は，$op = read$ のときは読み出された値，$op = write$ のときは書き込んだ値であり，オブジェクトあるいはプリミティブ型名を格納する．これらの属性値は，フィールドあるいは配列へのアクセスの前後に記録された情報から抽出される．

この抽出処理の例を示すため，実行履歴の例を**図 1** に示す．この図は，縦軸を時間軸，横軸をオブジェクトとしたもので，オブジェクト間でのコンストラクタ（`<init>`）やメソッドの呼び出しと，メソッドからの戻りを水平方向の矢印で示している．また，“`o2.list=o3;`” のような記述は，その時点でフィールドへの代入が実行されたことを示している．左側に示された $t_1, t_2, \cdots$ という並びは，縦軸での各位置に該当する時刻を意味するラベルである．この実行履歴を $call$ と $access$ の列に変換したものが**図 2** である．たとえば $t_{12}$ に実行された $o_2$ の `add` メソッドから $o_3$ の `add` メソッドへの呼び出しは，引数が $o_4$ であり，戻り値として boolean 型の値を返し，$t_{13}$ に終了したことを示している．

### 3.2 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．オブジェクト生成関係を形式化するにあたって，まず，次の 2 つの関数を定義する．

$call(t_1, t_5, o_1,$ `prepare` $, o_2,$ `<init>` $, \{\}, return, void)$
$call(t_2, t_3, o_2,$ `<init>` $, o_3,$ `<init>`$, \{\}, return, void)$
$access(t_4, write, field, o_2,$ `<init>` $, o_2, o_3)$
$call(t_6, t_{10}, o_1,$ `prepare` $, o_4,$ `<init>` $, \{\}, return, void)$
$call(t_7, t_8, o_4,$ `<init>` $, o_5,$ `<init>`$, \{\}, return, void)$
$access(t_9, write, field, o_4,$ `<init>` $, o_4, o_5)$
$call(t_{11}, t_{14}, o_1,$ `prepare` $, o_2,$ `add` $, \{o_4\}, return, void)$
$call(t_{12}, t_{13}, o_2,$ `add` $, o_3,$ `add`$, \{o_4\}, return, boolean)$
$call(t_{15}, t_{17}, o_1,$ `prepare` $, o_6,$ `<init>` $, \{o_2\}, return, void)$
$access(t_{16}, write, field, o_6,$ `<init>` $, o_6, o_2)$

図 2 実行履歴の例から抽出されるモデル
**Fig. 2** The execution model for the example trace.

$Obj(v)$ 値 $v$ がオブジェクトであるとき真，それ以外のとき偽を返す．
$K(t)$ 時刻 $t$ よりも前のイベントに出現したすべてのオブジェクトの集合を意味する．

これらの関数を使って，値 $v$ が時刻 $t$ で新しいオブジェクトであれば真を返す関数 $New$，値 $v$ が既知のオブジェクトまたは null，プリミティブ値であれば真を返す関数 $Exist$ を次のように定義する．

$$New(v,t) = Obj(v) \wedge v \notin K(t)$$
$$Exist(v,t) = \neg New(v,t) \wedge v \neq void$$

上記の関数に加えて，以下の条件のいずれかに該当するメソッドの集合を，生成に関係するメソッド集合 $CM$ として定義する．

(1) コンストラクタ．
(2) 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．経験的な基準から，メソッド名の最初の単語が set, add, put, push, offer, append, insert, replace のいずれかであるメソッドを，このカテゴリに含める．
(3) コンストラクタや状態変更メソッドの実行中に生成処理の一部として実行されるメソッド．具体的には，同一クラスに定義されたコンストラクタや状態変更メソッドの中からのみ呼び出されうるメソッドを，このカテゴリに含める．

また，メソッド集合 $CM_{lib}$ を，$CM$ のうち，ライブラリに所属する，すなわち観測対象外となっているメソッド群を意味する部分集合と定義する．

あるオブジェクトは，他のオブジェクトによってデータ参照などが行われた時点で，初期化が完了していると考えらえる．そのため，ある 1 つのオブジェクトについて，コンストラクタの実行が完了した後，そのオブジェクトの $CM$ に属さないメソッドが 1 回でも呼び出されると，その時点でオブジェクトの生成が完了したと見なし，それ以降の時刻におけるそのオブジェクトに関するメソッド呼び出しは，生成関係の特定処理から除外する．

#### 3.2.1 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，ライブラリの中で作られるオブジェクトについては，コンストラクタ呼び出しを観測できない．そこで本手法では，コンストラクタ呼び出しが観測できなかったオブジェクトについては，イベントに初めて登場した時点で，オブジェクトが構築されたと見なす．`main` メソッドなど，処理を呼び出す側のコードが観測対象外である場合，TRIGGER を特定することが不可能な場合がある．このような場合には，UNKNOWN オブジェクトという形式上のオブジェクトが，そのオブジェクトを構築したと出力する．

静的オブジェクトは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初期化されるため，TRIGGER となるオブジェクトは存在しない．その他の実行履歴に登場したオブジェクトには，必ず 1 つ，メモリ上に作られた他のオブジェクトあるいは UNKNOWN オブジェクトが，TRIGGER として存在することになる．

メソッド呼び出しイベントから TRIGGER を抽出するルールを**表 1** に，データアクセスイベントから TRIGGER を抽出するルールを**表 2** にそれぞれ示す．全部で 9 個のルールがあるが，すべてのコンストラクタ呼び出しが記録された実行履歴が与えられた場合，必要なルールは T1 だけである．残るルールは内部の観測が不可能なライブラリへの対応となっており，T2 および T3 は他のメソッド呼び出しの引数や戻り値に新しいオブジェクトが出現した場合に，T4 および T5 は新しいオブジェクト自身が何らかのメソッドを実行するかフィールドのアクセスを行った場合に，T6 は新しいオブジェクトに対するフィールドアクセスが生じた場合に，T7，T8 および T9 は新しいオブジェクトへの参照がフィールドや配列の読み書きに登場した場合に，それぞれ対応する．T8 は，フィールドの値が新しいオブジェクトのとき，所有者がオブジェクトを生成したと解釈する．T9 は，配列オブジェクト自身が要素を生成することはありえないため，配列を生成したオブジェクトが要素も生成したと解釈する．どのルールが適用された場合であっても，本研究では，観測している実行履歴に最初に出現した時刻に，コンストラクタ呼び出しに相当する処理が実行されたと見なす．

**表 1** メソッド呼び出しイベント $call(t_{call}, t_{ret}, o_1, m_1, o_2, m_2, P, s, r)$ からの TRIGGER の抽出ルール

**Table 1** The rules to extract TRIGGER from a method call events $call(o_1, m_1, o_2, m_2, P, t, r)$.

| 番号 | 条件 | 生成される辺 |
|---|---|---|
| T1 | $New(o_2, t_{call})$ | $o_1 \xrightarrow{TRIGGER} o_2$ |
| T2 | $New(p_i, t_{call})$ | $o_1 \xrightarrow{TRIGGER} p_i$ |
| T3 | $New(r, t_{ret})$ | $o_1 \xrightarrow{TRIGGER} r$ |
| T4 | $New(o_1, t_{call})$ | $UNKNOWN \xrightarrow{TRIGGER} o_1$ |

ただし $P = \{p_1, \ldots, p_n\}$ である．

#### 3.2.2 BASE の特定

各イベントで利用されたオブジェクトが生成に利用されたかを判断することで BASE-A，BASE-U の特定を行う．2 つのオブジェクト $o_1$ と $o_2$ の間に以下の関係があるとき，オブジェクトは生成に利用されたと考える．

- $o_1$ への参照が $o_2$ のフィールドに代入された．
- $o_1$ のメソッドが $o_2$ に対して，名前が add や set などの単語で始まるメソッドを呼び，引数としてデータを渡した．
- $o_1$ のメソッドを $o_2$ が呼び出し，何らかの値を $o_2$ が取り出した．
- $o_1$ のフィールドを $o_2$ が読み出した．

このような考え方に基づいて，7 つのルールを作成した．メソッド呼び出しイベントから BASE を抽出するルールを**表 3**，データアクセスイベントから BASE を抽出するルールを**表 4** にそれぞれ示す．B1 は，何らかのメソッドを呼び出して新しいオブジェクトが返ってくれば，呼び出し対象のオブジェクトと引数が，新しいオブジェクトに関係したと考える．また，B2 では，ライブラリに所属する `add` などのメソッド呼び出しで，オブジェクトを生成しないものは，オブジェクトの初期化に関するものであると考える．B3 と B4 は，生成に関連したメソッドから，ライブ

**表 2** データアクセスイベント $access(t_{access}, op, d, o_1, m_1, o_2, v)$ からの TRIGGER の抽出ルール

**Table 2** The rules to extract TRIGGER from $access(t_{access}, op, d, o_1, m_1, o_2, v)$.

| 番号 | 条件 | 生成される辺 |
|---|---|---|
| T5 | $New(o_1, t_{access})$ | $UNKNOWN \xrightarrow{TRIGGER} o_1$ |
| T6 | $New(o_2, t_{access})$ | $o_1 \xrightarrow{TRIGGER} o_2$ |
| T7 | $op = write \wedge New(v, t_{access})$ | $o_1 \xrightarrow{TRIGGER} v$ |
| T8 | $op = read \wedge d = field \wedge New(v, t_{access})$ | $o_2 \xrightarrow{TRIGGER} v$ |
| T9 | $op = read \wedge d = array \wedge New(v, t_{access})$ | $o'_2 \xrightarrow{TRIGGER} v$<br>ただし $o'_2$ は $o'_2 \xrightarrow{TRIGGER} o_2$ を満たすオブジェクト． |

表 3 メソッド呼び出しイベント $call(t_{call}, t_{ret}, o_1, m_1, o_2, m_2, P, s, r)$ からの BASE の抽出ルール

**Table 3** The rules to extract BASE from $call(t_{call}, t_{ret}, o_1, m_1, o_2, m_2, P, s, r)$.

| 番号 | 条件 | 生成される辺 |
|---|---|---|
| B1 | $s = return \wedge New(r, t_{ret})$ | $o_2 \xrightarrow{BASE-U} r, p_i \xrightarrow{BASE-U} r$ |
| B2 | $s = return \wedge Exist(r, t_{ret}) \wedge m_2 \in CM_{lib}$ | $p_i \xrightarrow{BASE-A} o_2$ |
| B3 | $s = return \wedge Exist(r, t_{ret}) \wedge m_1 \in CM \wedge m_2 \notin CM$ | $o_2 \xrightarrow{BASE-U} o_1, p_i \xrightarrow{BASE-U} o_1$ |
| B4 | $s = return \wedge m_1 \in CM_{lib} \wedge m_2 \notin CM \wedge Obj(r)$ | $\mathrm{r} \xrightarrow{BASE-A} o_1$ |

ただし $P = \{p_1, \ldots, p_n\}$ である．

表 4 データアクセスイベント $access(t_{access}, op, d, o_1, m_1, o_2, v)$ からの BASE の抽出ルール

**Table 4** The rules to extract BASE from $access(t_{access}, op, d, o_1, m_1, o_2, v)$.

| 番号 | 条件 | 生成される辺 |
|---|---|---|
| B5 | $op = write \wedge Obj(v)$ | $v \xrightarrow{BASE-A} o_2$ |
| B6 | $op = read \wedge m_1 \in CM$ | $o_2 \xrightarrow{BASE-U} o_1$ |
| B7 | $op = read \wedge d = array \wedge New(v, t_{access})$ | $o'_i \xrightarrow{BASE-U} v, v \xrightarrow{BASE-A} o_2$ ただし $o'_i$ は $o'_i \xrightarrow{BASE-U} o_2$ を満たすオブジェクト． |

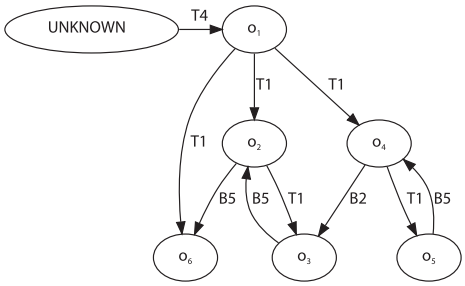


図 3 図 2 のモデルから得られるオブジェクト生成関係の例

**Fig. 3** A resultant graph for the trace model of Fig. 2.

ラリから外部へのコールバックを取り扱っている．B5 と B6 は，それぞれ，$o_1$ のフィールドに格納されたオブジェクトと，$o_1$ の生成途中で参照されるフィールドを持っていたオブジェクトからの関係を抽出する．最後に，B7 は，配列から未知のオブジェクト $v$ が読み出されたとき，$v$ が配列の初期化に使われたことを BASE-A として抽出する．$v$ の生成に関係するオブジェクト $o'_i$ が後から見つかれば，それらのオブジェクトから $v$ への関係を接続する．

なお，ルールに従って BASE を特定すると，任意のオブジェクト $A$ に対して，$A \xrightarrow{BASE} A$ という関係を特定する場合があるが，このような同一オブジェクトでの BASE は無視する．

図 2 に示した実行履歴のモデルから生成関係を抽出して得られるオブジェクト間の関係を**図 3** に示す．この図では，オブジェクトを頂点とし，辺のラベルに適用したルールを示している．オブジェクト $o_6$ からルール B2，B5 による BASE 関係をたどると，$o_4$ や $o_5$ が $o_6$ の生成に関係していることが読み取れる．この情報は，図 1 の呼び出し関係のみからでは読み取ることができない情報である．

### 3.3 オブジェクト生成関係の可視化

オブジェクトの生成関係は有向グラフであり，頂点はオブジェクトの情報として，パッケージ名，クラス名，オブジェクト ID の情報を保持する．また，辺は，TRIGGER および BASE の区別と，生成に関係したイベントの時刻を保持する．ここでのイベントの時刻とは，辺の生成条件に出現する，$t_{call}$，$t_{ret}$，$t_{access}$ のいずれかの値である．

本手法では，開発者が注目するオブジェクトを基点として，以下の基準で辺を逆向きにたどり，生成に関係するオブジェクトを抽出する．

- 基点から，BASE-A のみをたどって到達可能な範囲を探索する．
- BASE-U，BASE-A 辺を，直前にたどった辺の時刻よりも前の時刻のイベントから生成されたもののみ探索する．
- 上記の手順で到達した全オブジェクトから，TRIGGER 辺を 1 回だけたどる．

BASE-A は，オブジェクトに格納されたオブジェクトとなっているので，生成順序にかかわらず，すべてのオブジェクトが生成に関係したものと見なす．一方，BASE-U は，オブジェクトの生成に一時的に関係したものとなっているので，その時点でオブジェクトから参照可能であったものだけを探索する．探索によって発見された BASE オブジェクトに加え，各オブジェクトに対する TRIGGER オブジェクトを残した状態が，そのオブジェクトの生成に関係した範囲となる．

本研究では，提案手法のルールに基づいてグラフの可視化を行うツールを作成した．一般にオブジェクトは多数生成されるため，オブジェクト生成関係を閲覧する専用の

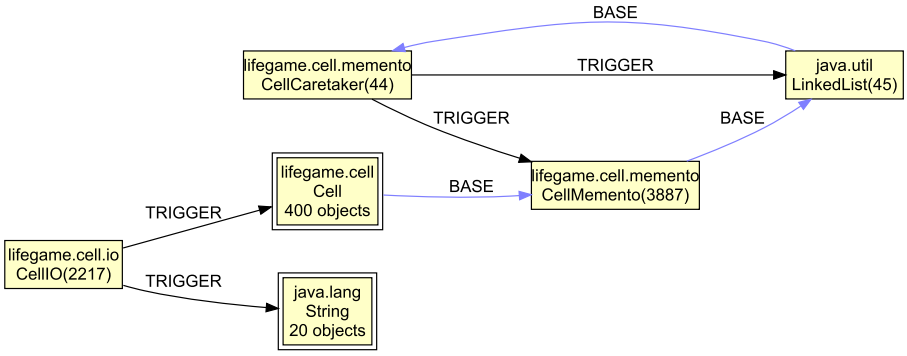


図 4 プログラム L3 の実行履歴から取得したオブジェクト生成関係の例
**Fig. 4** A resultant graph extracted from an execution trace of a program L3.

ツールとし，頂点の集約機能を実装した．この集約機能は，1 つのクラスのオブジェクト群が，ある 1 つのオブジェクトに対して同一ラベルの辺を持つとき，オブジェクト群を 1 つの頂点として表示する．また，オブジェクトを迅速にグラフ中から特定するために，クラス名，オブジェクト ID を用いた検索を可能とした．また，辺の表示では，基本情報として TRIGGER，BASE のラベルを表示し，開発者が選択した辺について，その辺に対応するイベント情報を閲覧できるようにした．

## 4. ケーススタディ

提案手法を実装したツールを用いて，プログラム理解のケーススタディと，ツールの性能評価を実施した．

ケーススタディの対象としたのは，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「コンウェイのライフゲーム」プログラムである．このプログラムは GUI を持ち，起動するとウィンドウに碁盤目状の「盤面」を表示する．利用者はマウスを使って，各区画「セル」の状態を編集したり，ライフゲームのルールを適用してセルの状態を更新したりできる．その他の機能として，盤面を過去の状態に巻き戻す機能や，盤面の状態をファイルに保存する機能を持つ．

プログラム理解に関するケーススタディでは，当該課題の内容を知らない第 1 著者が，課題の文書に書かれた外部仕様を読解したのち，3 名の学生が作成したプログラム L1，L2，L3 を分析し，グラフに可視化されているオブジェクトの生成方法の有用性を評価した．

対象となっている 3 つのプログラムは，同一の要求に基づいて作成されているが，作成者ごとに独自の機能が追加されている．すべての作成者は，設計の例として，基本機能に関するクラス分割および各クラスが持つべきメソッド名の情報を与えられており，L1 と L2 は，この設計例に従ったプログラム構造を持った学生のプログラムから，ランダムに選択したものとなっている．一方，L3 は，設計例に従っていないもののうち，最も複雑であることを期待して，一番ファイル数が多いものを選択した．プログラムのサイズは，空行やコメントも含めたとき，L1 は 9 クラス 594 行，L2 は 8 クラス 1,486 行，L3 は 52 クラス 2,780 行である．ここで，L2 は L1 に比べて非常に大きく見えるが，L2 では独自拡張として，ライフゲームにおける著名な盤面のパターンを表示する機能を組み込んでおり，パターンに対応した盤面のデータを約 500 行にわたってハードコーディングしているため，実際の機能量に比べて大幅にサイズが大きくなっていた．また，複数行にわたるコメントおよび空行をレイアウトに数多く使っていたため，実際の有効なコードの量は，L1 とほぼ同程度である．L3 は L2 と比べてもサイズが大きいが，L3 の作者は 1 クラスを 1 機能とする形でクラス分割を行った結果，たとえばファイルを読み込むクラスや，それを継承した設定ファイルを読み込むクラス，ライフゲームの盤面を読み込むクラス，といったようにクラスの定義を行っており，多くのコードをクラス宣言に費やしている．

### 4.1 実行シナリオの分析

まず，クラス数が多く特殊な構造を持つとみられる L3 について，提案手法を用いて，盤面データをファイルから読み込み機能の実行を分析した．調査方法であるが，まず，L3 を起動し，盤面の状態を保存しているファイルを開いてから，L3 を終了するというシナリオを実行し，実行履歴を取得した．この実行履歴に対して，第 1 著者が提案手法を適用し，オブジェクトの生成処理を可視化したグラフを取得した．図 4 は実行履歴を解析し，盤面の状態を管理するオブジェクト CellCaretaker の生成処理の一部を可視化したものである．プログラム L3 のファイルの読み込み機能の実行では，まず，CellIO オブジェクトがファイルの内容を格納した String オブジェクトを生成する．次に，CellIO は，文字列の内容を読み込み，盤面の各セルの状態を表す Cell オブジェクトの集合を生成する．CellCaretaker オブジェクトは，Cell オブジェクトの集合から，盤面の状態を表現する CellMemento オブジェクトと，盤面の履歴を表現する LinkedList オブジェクトを生成し，盤面の状態が 1 つだけ登録された状態となる．

作業を行った第1著者は，提案手法によって可視化されたグラフから，図4に相当する部分を見つけ，CellIOがCellの生成を，CellCaretakerがCellMementoの生成を実行していること，Cellのデータが CellMementoに格納されていることを推測した．その後，推測が正しいことを，CellCaretaker，CellIO，Cell，CellMementoの4つのクラスのソースコードを読解することで達成した．一方で，提案手法を使わない調査方法の代表として，デバッガによるステップ実行で，ファイル読み込み機能の実行開始（メニューの `actionPerformed` メソッド）から，ファイル読み込みを追跡した．この方法では，ファイル読み込みの開始点となったOpenListenerクラスから，盤面情報を読み込むCellIOクラスの処理に至るまでに，ファイルを開くクラス，ファイルのヘッダを読み込むクラスなど，合計7つのクラスのコードを通過した．これは，L3では共通処理をできるだけ継承によって親クラスにまとめるようにコードが記述されており，読み込んだデータが画面に反映されるまでに，全部で25個のクラスのコードが実行された．提案手法は，ファイルからの盤面の読み込みに無関係なGUI，ファイルIOなどのコードを提示しないため，単純なデバッガによる処理の追跡などに比べると，簡潔な情報を提示できたといえる．

### 4.2 メソッドのテスト用データの作成

オブジェクトの生成に関して十分な情報が可視化されているかを確認するために，ライフゲームの実装L1とL2に対して，可視化されたグラフの情報から，メソッドの単体テストに必要な状態のオブジェクトを生成するコードを記述する作業を実施した．L3は，すでに4.1節で分析しているので，対象から除外している．

デバッガや保守におけるプログラム理解に役立つオブジェクト生成関係とは，ある処理を実行するために必要なオブジェクトの情報，すなわち，メソッドを実行するオブジェクトや，その時点で引数やフィールドを介して参照可能なオブジェクトの状態に関する情報である．グラフがそのような情報を含んでいるのであれば，グラフに記述された情報をもとに，任意のメソッドについて，そのメソッドの単体テストを実施可能な状態のオブジェクトを生成するソースコード記述が可能であると考えた．単体テストを実施するためには，実行時エラーを起こさず，かつ，十分なカバレッジを達成できるようにオブジェクトを生成しなくてはならない．テスト対象メソッドごとに，ホワイトボックステストの観点から，条件分岐がすべて網羅できれば，そのメソッドのテストが成功していると判定する．

テスト対象にしたクラスは，1枚の盤面の状態を表現するBoardModelクラス，盤面の履歴を管理するBoardHistoryクラス，ライフゲームの進行を管理するGameControllerクラスの3つである．授業の指導書に記載されたクラスの情報から，他のクラスはGUIの実装に対応していたため，テスト対象から除外した．BoardModeクラスは他のクラスに依存しておらず，BoardHistoryはBoardModelクラスに，GameControllerクラスはBoardHistoryクラスに依存していることから，BoardModel，BoardHistory，GameControllerの順でテストを作成するものとした．なお，これらのクラスには，授業の指導書に記載のない，学生がそれぞれ追加したメソッドも含まれている．

グラフからソースコードを作成する作業は，第1著者が実施した．第1著者の能力が結果に影響するのを避けるため，プログラムの実行履歴は，授業での動作確認手順を記載した手順書に従って，プログラムを起動し，実装が必須とされていた機能をひととおり実行することで記録した．また，グラフに含まれた情報の有効性を評価するため，テストを作成する手順では，オブジェクトの生成としてソースコードに記述してよいメソッド呼び出しを，グラフ上で各クラスのオブジェクトの生成からたどれる範囲に表現されている呼び出し関係に限定し，引数として使用できるプリミティブ型の値は，現在テスト対象となっているクラスのソースコードに定義されている値，そのクラスのコンストラクタ呼び出しにハードコーディングされている値に限定した．

本実験の結果をまとめたものを**表5**に示す．表5において，メソッド数とは，テスト対象となったpublicメソッドの数である．平均LOCとは，テスト用メソッドにおける，空行，コメント，テスト用 `assert` 文を除いたコード行数の平均である．テスト対象メソッドを実行するために必要な事前準備や，テストすべき値を取り出す処理が長いほど，この値が大きい．テスト成功メソッド数とは，ホワイトボックステストとしてテスト用コードを評価したとき，各テスト用コード単独で条件分岐を網羅できるようにオブジェクトを準備できたメソッドの数である．対象とした61メソッド中46メソッドのテストについて，可視化されたグラフの情報と，対象クラスのソースコード情報を組み合わせるだけで，必要なオブジェクトの生成コードを記述することができた．また，Cobertura[*1]を用いてすべてのテストをまとめて実行し，行カバレッジおよび分岐カバレッジを計算したところ，L1に関しては行カバレッジおよび分岐カバレッジは100%，L2に関しては行カバレッジ99%，分岐カバレッジ87.5%となっていた．L1に関してはテストメソッドの作成成功の判断に比べ数値が向上しているが，これは，BoardModelの単独のテストでは実行できなかった処理が，GameControllerなどのテストにおいて実行されたためである．L2については，GameController内部に，GUIでのアニメーション機能に関する条件分岐があり，その範囲についてはテスト不可能であったため，カ

[*1] Cobertura. http://cobertura.sourceforge.net/

表 5 メソッドのテスト用オブジェクトの生成結果
**Table 5** Result of generating essential objects for method tests.

| 対象 | クラス | メソッド数 | 平均 LOC | テスト成功メソッド数 | 割合（%） |
|---|---|---|---|---|---|
| L1 | BoardModel | 7 | 8.57 | 3 | 42.86 |
| L1 | BoardHistory | 7 | 3.71 | 7 | 100.00 |
| L1 | GameController | 12 | 5.50 | 12 | 100.00 |
| L2 | BoardModel | 10 | 9.50 | 4 | 40.00 |
| L2 | BoardHistory | 8 | 3.75 | 7 | 87.50 |
| L2 | GameController | 17 | 26.88 | 13 | 76.47 |

バレッジは 100%に到達しなかったが，GUI に関係ない部分はすべて実行されていた．結果をまとめると，単体テストとして，各テスト用メソッドでテスト対象メソッドの条件分岐を網羅できたものは 61 メソッド中 46 メソッドであり，テストケース集合全体としては，GUI と無関係に単体テスト可能なコードについては，網羅することに成功していた．

GameController クラス，BoardHistory クラスは，盤面の状態を管理するクラスとなっていることから，少なくとも 1 つ，ライフゲーム開始時の盤面（BoardModel オブジェクト）の登録が必要であり，そのメソッドを呼び忘れると，いくつかのメソッドは正しく動作しない．また，テスト対象の 3 つのクラスにおいて，L1 は 6 個，L2 は 15 個のメソッドを独自に実装しており，これらの中には，複数の盤面の状態が履歴として蓄えられたときに初めて実行可能となる（盤面がたかだか 1 つしかない状態で呼び出すとクラッシュする）メソッドが存在していた．これらのメソッドの実行に必要なオブジェクトの生成方法を，外部仕様なしでもグラフから抽出できたことから，オブジェクトの生成に関する十分な情報をグラフが表現していたと考えられる．

なお，テストが成功しなかったメソッドについては，実行履歴から作成したグラフと，テスト対象クラスのソースコードのみでは，プリミティブ型の引数の値などに適切なものを用意できず，作成することができなかった．たとえば，盤面のある座標にあたるセルが生存しているか，していないかを判定するメソッドをテストするには，セルの座標の値が必要であるが，テスト対象コードの範囲では，その値が提供されていなかった．この点は本研究の実装上の制約に由来しており，プリミティブ値を記録するように実装を拡張すれば対応可能である．

### 4.3 実行性能の調査

ツールの実行性能を，Intel Xeon(R) E5507 2.27 GHz のワークステーション上で計測した．解析対象は，Ant と ANTLR の既存のテストケースの実行と，Java の実行ベンチマーク集 DaCapo の各ベンチマークである．実行履歴の記録の際に正常に動作しなかった（テストそのものに失敗した）ものを除外し，56 の実行履歴に対して調査を行った．

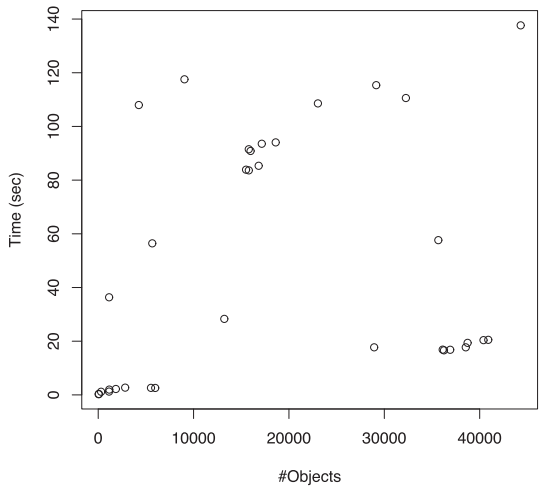


図 5 オブジェクト数に対する生成関係特定時間の関係
**Fig. 5** Time to identify relationship of object generation.

実行履歴に出現したイベント数は最大で 2,500 万，オブジェクト数は 30 万であった．実行時オーバヘッドは，主として実行履歴をディスクに保存するために要する時間であるが，最も大きいもので 2 分であった．

イベント数とオブジェクト数が，それぞれ生成関係の特定に必要な時間に影響する度合いを調べた結果，イベント数と計算特定時間の相関係数が 0.45，オブジェクト数と特定時間の相関係数が 0.93 となった．このことから，オブジェクトの数が生成関係の特定に要する時間と強く関係している．解析対象となるオブジェクトの数は，解析の範囲をたとえば 1 つのメソッドの実行開始から終了までというように区切れば，容易に制御することができる．

本手法では，1 度にオブジェクト生成関係を特定するオブジェクト数は数万程度を想定している．オブジェクト数が 5 万以下の実行履歴において，オブジェクト生成関係の特定に必要な実行時間を調査した結果を**図 5** に示す．この図から，半分以上の実行履歴で 1 分以下，ほとんどの実行履歴において 2 分以下でオブジェクト生成関係を特定しており，実行時オーバヘッドまで含めても 5 分程度となったことから，現実的な時間で利用可能なツールであるといえる．

### 4.4 議論

ケーススタディでは，第 1 著者が提案手法を用いて，プログラム理解および単体テストの作成を行った．対象プログラムは第 1 著者は初めて見るプログラムであり，実行履歴は授業課題の動作確認用に事前に定義された操作手順を用いている．また，ファイルとして開いてよいソースコードを，テスト対象クラスおよびテストが完了したクラスに制限し，テスト記述に使用してよい情報をグラフに示された情報に限定することで，グラフの情報をソースコードに手作業で変換するという形式をとり，第 1 著者の推測などが結果に影響しないように実施した．そして，結果の評価は，単体テストにおいて，メソッドを呼び出すために必要なオブジェクト生成を行うことができたこと，ホワイトボックステストとしての妥当性，行カバレッジと分岐カバレッジによって評価した．L1 と L2 に関する結果から，グラフを閲覧することでオブジェクトの生成に関する情報を得ることができることを示している．また，L3 に関する結果から，デバッガによるステップ実行に比べて，データフローに関与する少数のクラスだけを迅速に特定できる場合があることを示した．

この有効性の評価方法は，オブジェクトの関係を可視化する関連研究 [2], [3], [8] と同様に，著者によるケーススタディによる有効性の提示という方式を採用している．プログラム可視化手法がプログラム理解に与える影響を直接評価することは難しいため [12]，ケーススタディでは，オブジェクトの使用方法をグラフから読み取ることが可能である，という情報の含有を評価の基準としている．この可視化が，異なる作業目的ごとのソースコード読解作業の時間をどれだけ効率化するか，という時間面での定量的な評価は今後の課題とする．

なお，提案手法は，プログラムの実行履歴全体を可視化するため，本ケーススタディのように，特定のタスクの実施に必要な情報を含んでいることは示すことが可能であるが，正確さに関しては，他の情報をすべて余計な情報と解釈すると，正確さは実行履歴のサイズによって定まってしまうため，本研究では議論していない．また，ライブラリや Java 仮想マシンの内部で生じるイベントを含めた完全な生成関係は観測することができないため，それに対する正確さも本研究では議論の対象外とした．

## 5. 関連研究

本研究は，オブジェクトの生成関係を動的解析によって抽出する手法を提案した．動的解析の分野では，オブジェクトの呼び出し関係，所有関係，参照関係を可視化する手法が複数提案されている．まず，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 [9]．この手法は，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ソッド呼び出しを時系列で示すことに主眼を置いているため，図 1 に示した例のように，オブジェクトが引数に含まれていた場合には，オブジェクト間の関係を読み取れない場合がある．

Rayside ら [8] は，動的解析によって，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，所有関係を特定し可視化する．この手法は，所有関係を可視化するが，オブジェクトの構築段階において，一時的にデータを参照する関係については可視化できず，どのようにオブジェクトが生成されたかを知りたい場合には使用することができない．

Lienhard ら [2] 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．また，Lienhard ら [3] は，この手法を利用して，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．これらの手法は，オブジェクトがどのオブジェクトにアクセスですることが可能であったかを提示し，メソッドの実行によって生じた状態変化を知ることができる．本研究では，時系列によって，その時点で実際に利用されたオブジェクトだけを提示しており，開発者に提示する情報を減らしているといえる．

オブジェクトの生成方法を理解するという観点で，2 つの静的解析手法が提案されている．Mandelin ら [6] は，利用者が指定した条件を満たすメソッド呼び出し列を生成し，提示する手法を提案している．また，Thummalapenta ら [10] は，ソースコード検索エンジンを用いて集めたコードサンプルを利用し，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．これらの手法は静的に行われているため，取得するオブジェクトの状態までは完全には再現されないことがある．これらの手法は，複数のプログラムを同時に解析できる点で有意義であるが，本研究のように，ある特定のクラスの生成を理解するという用途とは目的が異なっている．

## 6. まとめ

本研究では，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特定し，有向グラフとして可視化する手法を提案した．ケーススタディでは，プログラムの特定の機能を分析する際の実例を示し，また，メソッドの単体テストに使用できるようなオブジェクトの生成方法を，グラフから読み取ることができることを示した．

今後の課題としては，プログラム理解への有効性を確認する対照実験があげられる．また，実行履歴の取得範囲が制限されている環境下での動的解析手法について，本研究で使用したルールの一般化を試みる予定である．

**謝辞** 本研究は科研費（23680001）の助成を受けたものである．

## 参考文献


[1] LaToza, T.D. and Myers, B.A.: Developers Ask Reachability Questions, *Proc. 32nd ACM/IEEE International Conference on Software Engineering*, pp.185–194 (2010).

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

[3] Lienhard, A., Girba, T., Greevy, O. and Nierstrasz, O.: Test Blueprints – Exposing Side Effects in Execution Traces to Support Writing Unit Tests, *Proc. 12th European Conference on Software Maintenance and Reengineering*, pp.83–92 (2008).

[4] Lo, D. and Maoz, S.: Scenario-based and value-based specification mining: Better together, *Proc. 25th IEEE/ACM International Conference on Automated Software Engineering*, pp.387–396 (2010).

[5] 前田直人：コンテキストを考慮したポインタ解析結果を用いたメソッド呼出しパターン検査，コンピュータソフトウェア，Vol.26, No.2, pp.157–169 (2009).

[6] Mandelin, D., Xu, L., Bodik, R. and Kimelman, D.: Jungloid Mining: Helping to Navigate the API Jungle, *Proc. ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, pp.48–61 (2005).

[7] 宗像 聡，石尾 隆，井上克郎：類似する振舞いのオブジェクトのグループ化によるクラス動作シナリオの可視化，情報処理学会研究報告，第 163 回ソフトウェア工学研究発表会，Vol.31, pp.225–232 (2009).

[8] Rayside, D., Mendel, L. and Jackson, D.: A Dynamic Analysis for Revealing Object Ownership and Sharing, *Proc. 2006 International Workshop on Dynamic Analysis*, pp.17–23 (2006).

[9] 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).

[10] Thummalapenta, S. and Xie, T.: PARSEWeb: A Programmer Assistant for Reusing Open Source Code on the Web, *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, pp.204–213 (2007).

[11] Weiser, M.: Program slicing, *Proc. 5th International Conference on Software Engineering*, pp.439–449 (1981).

[12] Wettel, R., Lanza, M. and Robbes, R.: Software Systems as Cities: A Controlled Experiment, *Proc. 33rd International Conference on Software Engineering*, pp.551–560 (2011).



**中野 佑紀**

平成 23 年大阪大学大学院情報科学研究科博士前期課程修了．同年株式会社ニッセイコム．在学時，プログラムの動的解析の研究に従事．



**伊達 浩典** （学生会員）

平成 19 年関西大学総合情報学部総合情報学科卒業．平成 21 年大阪大学大学院情報科学研究科博士前期課程修了．現在，大阪大学大学院情報科学研究科博士後期課程在学中．ソフトウェアに対するパターンマイニングの研究に従事．



**渡邊 結** （正会員）

平成 20 年大阪大学大学院情報科学研究科博士前期課程修了．平成 23 年同研究科博士後期単位取得退学．同年日立製作所研究開発本部横浜研究所．在学時，プログラムの動的解析の研究に従事．



**石尾 隆** （正会員）

平成 15 年大阪大学大学院基礎工学研究科博士前期課程修了．平成 18 年同大学院情報科学研究科博士後期課程修了．同年日本学術振興会特別研究員（PD）．同年ブリティッシュコロンビア大学ポストドクトラルフェロー．平成 19 年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助教．博士（情報科学）．データフロー解析，アスペクト指向プログラミングに関する研究に従事．日本ソフトウェア科学会，ACM，IEEE 各会員．



**井上 克郎** （フェロー）

昭和 54 年大阪大学基礎工学部情報工学科卒業．昭和 59 年同大学大学院博士課程修了．同年同大学基礎工学部情報工学科助手．昭和 59〜61 年ハワイ大学マノア校情報工学科助教授．平成元年大阪大学基礎工学部情報工学科講師．平成 3 年同学科助教授．平成 7 年同学科教授．平成 14 年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教授．平成 20 年国立情報学研究所客員教授．同年情報処理学会フェロー．同年電子情報通信学会フェロー．工学博士．ソフトウェア工学の研究に従事．日本ソフトウェア科学会，電子情報通信学会，IEEE，ACM 各会員．