プログラム実行履歴からの簡潔なシーケンス図の生成手法

谷口 考治　石尾 隆　神谷 年洋　楠本 真二　井上 克郎

オブジェクト指向プログラムの動作を理解するためには，生成されるオブジェクト群がどのようにメッセージ通信を行うかを理解する必要がある．しかし，オブジェクトの動作は動的束縛などによって動的に決定されることが多く，静的な情報であるソースコードからオブジェクトの動作を理解することは困難である．本研究では，Unified Modeling Language (UML) のシーケンス図に着目する．プログラムの実行履歴を基にシーケンス図の作成を行うことで，プログラム実行中に生成される各オブジェクトがどのように動作するのかを視覚的に示す．一般に実行履歴は膨大な量に上るため，作成されるシーケンス図は非常に大きくなる．この問題に対し，本研究では，実行履歴中から繰り返しや再帰構造になっている部分を検出し，簡潔な表現に置き換えることで，提示する情報量を削減する手法を提案する．

A software system developed by object-oriented programming operates by message exchanges among the objects allocated by the system. To understand such system behavior we need to grasp how the objects are communicating. However, it is difficult to understand the behavior of such system, since it has many elements determined dynamically, and many objects are usually related to one functionality. We propose a method of extracting a sequence diagram from an execution trace of a Java program in order to understand the behavior of the program. Our method compresses a repetition included in the execution trace. The compressed execution trace is shown as a sequence diagram. This paper presents four compression rules designed for object-oriented program. The experiment illustrates how our rules effectively compress the execution trace and extract a sequence diagram from the result.

1 はじめに

近年，ソフトウェア開発によく用いられるオブジェクト指向技術で作成されたプログラムでは，実行時に動的に生成されるオブジェクトが相互にメッセージを交換することによってシステム全体が動作する．どのオブジェクトがどのようなタイミングでメッセージ通信を行うかは，実行時に動的に決定される．そのため，設計や実装作業においても常に，システムが生成するオブジェクトの動的な振る舞いをイメージしながら作業を進めなければならない．保守過程において，プログラムの動作を理解する際も同様である．保守作業者は，保守対象のプログラムの動作を理解しようとするとき，プログラムの静的な記述であるソースコードを見ながら，実行時に生成されるオブジェクト群の動作をイメージして，理解していかなければならない．しかし，これは非常に困難な作業であり，動的束縛などを伴う複雑なオブジェクト群の動作や関連性を理解することは難しい[5][14]．そのため，生成され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手法が求められている[8][9]．

Generating Compact Sequence Diagram from Execution Traces.

Koji Taniguchi, Shinji Kusumoto and Katuro Inoue, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, Osaka University.

Takashi Ishio, 大阪大学大学院情報科学研究科, Graduate School of Information Science and Technology, 日本学術振興会特別研究員 (PD), Research Fellow of the Japan Society for the Promotion of Science.

Toshihiro Kamiya, 独立行政法人産業技術総合研究所, National Institute of Advanced Industrial Science and Technology.

コンピュータソフトウェア, Vol.24, No.3 (2007), pp.153–169.
[論文] 2005 年 12 月 28 日受付.

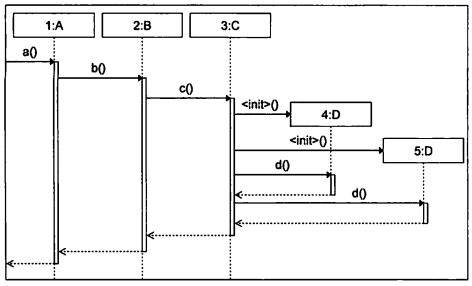

図 1 シーケンス図

オブジェクトの動作を表現する方法として，UML [13] のシーケンス図 (図 1) がある．シーケンス図とは，時間軸に沿って，オブジェクトがどのような順序でメッセージ通信を行うかを表現する図である．シーケンス図は通常，設計時に作成される．しかし，全てのプログラムについてシーケンス図が作成されている訳ではなく，また，作成されていたとしても，設計時に作成された図と実際のプログラムの動作とでは，異なっていることがある．そこで，我々は，プログラムを解析し，その結果を基にシーケンス図を作成することで，実際のプログラムの動作を図示する．この図を用いることで，実際にプログラム中で作成されるオブジェクトがどのようにメッセージ通信を行うかを理解することができる．また，設計段階で作成されたシーケンス図と，実装されたプログラムの振る舞いとの違いを調べることや，特定のバグを再現させるテストケースのシーケンス図と，バグを再現させないシーケンス図を比較することで，本手法をデバッグ作業に利用することなども想定している．

実行時のオブジェクト間の動作を正確に解析するために，プログラムの動的解析を行い，その結果として得られた実行履歴を基に図の生成を行う．しかし，一般的に，プログラムの実行履歴は膨大な量になる [9] [10] [11]．そのままシーケンス図にしても大きな図ができあがってしまい，理解するのは難しくなる [4]．そのため，できる限り情報量を削減しなければならない [9] [11]．

そこで，我々は，実行履歴を圧縮し，全体の情報量を削減する手法を提案する．具体的には，実行履歴中からループや再帰構造によって発生する繰り返し構造を検出し，それらを抽象化して簡潔な表現に置き換える操作を行う．我々は，圧縮手法として，オブジェクト指向プログラムの構造を考慮した 4 つの実行履歴圧縮ルールを考案した．また，その結果を基にシーケンス図を作成し，繰り返しになっている部分などを注釈として図中に表現する表記法についての提案も行う．これらの手法を用いることで，プログラム全体の動作を表現する簡潔なシーケンス図の作成を行うことができる．後述する適用実験では，数十万回のメソッド呼び出しを含む実行履歴を，提案手法を用いることで 14%から 0.8%程度まで圧縮することができた．

また，実行履歴の取得，提案する圧縮ルールの適用，シーケンス図の作成，作成したシーケンス図の閲覧，操作ができるツールの実装を行った．このツールでは，圧縮した箇所を任意に展開し，元の詳細な情報を取り出せるようにしている．この機能を用いて，圧縮した図から全体の流れの概要を理解し，さらに個別に詳細な振る舞いを見たい箇所を展開して理解することができる．

以降，2 章では圧縮手法の詳細と，圧縮結果のシーケンス図への表記法について述べる．また，3 章では適用実験の結果を述べる．4 章では提案手法に関する考察を行う．5 章では関連研究について，6 章でまとめと今後の課題について述べる．

2 実行履歴圧縮手法

本章では，本手法で使用する実行履歴とその圧縮手法の詳細，および，圧縮結果を基にしたシーケンス図の作成手法について述べる．

2.1 実行履歴

本手法では，まず，対象とするプログラムを実行し，実行中に発生するメソッド呼び出しに関する情報を実行履歴として取得する．

実行履歴情報としては，個々のメソッド呼び出しについて次のような情報を仮定している．

- 呼び出しを受けたオブジェクトのオブジェクト ID
- 呼び出されたメソッドのメソッド名，引数の型，返り値の型
- 呼び出されたメソッドを実装しているクラス名

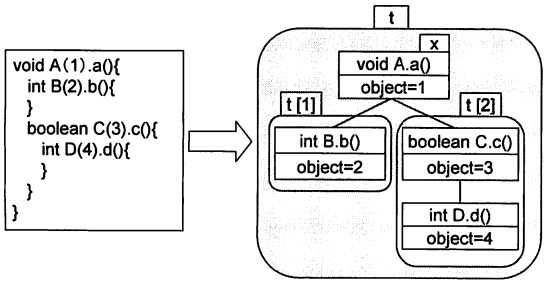

図 2 木構造と記法の例

- メソッドの開始，終了のタイミング

また，static メソッドの呼び出しの際には，オブジェクト ID を 0 と考える．継承などによる多態性がある場合には，動的束縛によって実際に呼び出されたメソッドとそれを実装しているクラス名を記録しているものとする．これらの情報から，実行時のメソッド呼び出し構造を再現することができる．

以降，本稿において実行履歴中の情報を表現する際には主にテキスト形式で表現したものを用いる (図 2 の左図)．この表現中では，各行がメソッドの開始，または終了を表し，メソッドの開始を表す行では，前述した内容を「**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){**」，という形式で記述する．メソッドの終了を表す行には閉じ括弧「**}**」を記述する．また，メソッド呼び出しの入れ子構造を字下げを用いて表現する．

2.2 圧縮アルゴリズムの設計方針

本研究は，実行時のメソッド呼び出し情報を，簡潔な形式でシーケンス図上に表現することで，実行時の動作を利用者が理解しやすくすることを目的としている．そのため，本研究における実行履歴の圧縮アルゴリズムは以下の条件を考慮して設計しなければならない．

1. シーケンス図上に，圧縮の適用結果が簡潔に図示できる必要がある
   最終的にシーケンス図上に表現することを目的とした圧縮であるため，圧縮結果がシーケンス図上に図示可能な形態をしている必要がある．また，利用者はシーケンス図上に表示された圧縮結果を見て対象プログラムの動作概要を理解するため，圧縮された箇所が図上で判別できること，圧縮結果を人間が見て，理解できる構造になっていることが必要である．
2. 圧縮の適用前の振る舞いの推測が容易である
   シーケンス図上に表示された圧縮結果を見て圧縮前の処理の構造が推測できなければならない．例えば，繰り返し発生しているメソッド呼び出しをループ構造として圧縮した場合，圧縮結果に繰り返し回数等を記録し図上に表示しておく．こうすることで，その部分が圧縮処理を受けており，実際には同じメソッド呼び出しがその回数だけ繰り返されているという構造が推測できる．しかし，圧縮結果から圧縮前の状態を推測した結果が誤っている場合，その部分の実行時動作を誤って理解してしまうことになる．このようなことを防ぐために，圧縮結果は紛らわしくない単純な表現で図上に表現できるようにし，誤った理解を与えないようにすることも重要である．
3. 特定の部分だけを局所的に選んで圧縮できる必要がある
   実行履歴の任意の一部分のみを圧縮する，または任意の一部分のみを圧縮しないというように，局所的に圧縮状態を制御できることが望ましい．こうすることで，詳細な情報を表示する箇所と，概要情報として圧縮結果を表示する箇所に分けて図上に表示することができる．具体的には，デバッグ実行中にブレークポイント等で停止した際に，ブレークポイントに近い箇所は詳細を表示し，遠い部分については圧縮した概要情報を表示する，というような利用方法が考えられる．
4. 圧縮前の状態に戻せるようなアルゴリズムにする必要はない
   圧縮を行うことで圧縮前の状態よりも情報量が削減されるため，本手法の利用者は任意の部分に対して，より詳細な圧縮前の実行系列を参照したくなることが考えられる．そのため，圧縮された箇所について，任意に元の情報を参照できるようにする必要がある．しかし，本手法による圧縮は提示する情報を削減することが目的であり，保存容量の削減が目的ではない．そこで，非可逆

な圧縮アルゴリズムであっても，圧縮前の状態も保存しておき，必要に応じて表示する情報を入れ替え圧縮前の情報を提示すればよい．条件 3 を満たしていれば圧縮処理は局所的に圧縮状態と，非圧縮状態を選択できるようになっているため，局所的に元の情報を提示することが可能である．よって，圧縮アルゴリズムとしては可逆性を考慮する必要はない．

2.3 圧縮アルゴリズム

2.1 節で述べた実行履歴中には，プログラム実行中に発生するメソッド呼び出しが全て記録されているため，実行履歴は膨大な量に上ることが多い [10]．そこで我々はプログラムのループ構造や再帰構造に着目する．これらの構造は繰り返し実行することを目的として作られているため，そこから発生するメソッド呼び出しの実行履歴には同じようなメソッド呼び出しが繰り返し出現していると考えられる．このようなメソッド呼び出し構造を元のループ構造や再帰構造の形式に戻し，シーケンス図上に表現することで，シーケンス図を簡略化することができる．

ループ構造や再帰構造はソースコード上で一般的に用いられている形式であるため，メソッド呼び出し系列をその構造を用いて表現することが可能である．シーケンス図上であっても，ループや再帰呼び出しを表す簡単な注釈を書き加えることで，その構造は容易に図示することができる (条件 1)．また，ソフトウェア開発者にとっては身近な形式であるため，その構造の意味の理解や，実際の動作の推測が容易である (条件 2)．ただし，メソッド呼び出し系列をループ構造や再帰構造を用いて表現するといっても完全にソースコードと同様の形式にできるわけではない．なぜなら，ソースコード上の記述では，呼び出されたメソッドの内部までは記述されないのに対して，シーケンス図は，あるメソッド呼び出しの中から発生する全てのメソッド呼び出しを含んだメソッド呼び出し構造を表現するからである．そのため，繰り返し構造の圧縮においては，単純にメソッド呼び出しが繰り返されている箇所を検出するのではなく，メソッド呼び出し構造全体が繰り返されている箇所を検出し，その繰り返し 1 回分のメソッド呼び出し構造を 1 つの単位として圧縮しなければならない．そこで我々は，実行履歴を，メソッド呼び出し構造を表現した木構造として扱い，その中の任意の部分木を単位としてループや再帰構造を形成する．また，木構造として扱うことで，木全体ではなく一部の部分木に対しての圧縮処理が可能となり，局所的な圧縮を行うことが可能である (条件 3)．さらに，木構造であるため，実行履歴上の「メソッドの開始」イベントと「メソッドの終了」イベントの対応関係や順序関係を崩さないように圧縮することが容易になるという利点もある．以上のことから，実行履歴上のメソッド呼び出し構造を木構造で表現し，その中に含まれる繰り返しをループ構造や再帰構造の形式を用いて表現することは 2.2 節で述べた条件を満たしている．

以下，実行履歴のメソッド呼び出し構造を表現した木構造について説明する．図 2 は木構造の例である．この木構造では，各メソッド呼び出しを節点とし，それぞれの節点は，呼ばれたメソッドのシグネチャと呼び出しを受けたオブジェクトの ID を持つ．また，あるメソッド呼び出しの内部で呼ばれるメソッド呼び出しを，その節点の子として辺を引く．子節点の順序は対応するメソッドが呼び出される順番である．例えば，図 2 の左側のような実行履歴は，右側の木構造に置き換えられる．次に，表 1 に示す記法を定義する．図 2 において，実行履歴中の最初のメソッド呼び出しである，オブジェクト ID1 番のオブジェクトに対するクラス A のメソッド a() の呼び出しを表す節点を x とする．x.method はクラス A のメソッド a() を指し，x.object はオブジェクト ID1 番のオブジェクトを指す．このとき，このメソッド a() の呼び出しの中で発生するメソッド呼び出しは，メソッド b() の呼び出しとメソッド c() の呼び出しの 2 つであるから x.callNum は 2 である．また，t を x を根とする木構造とすると，t の子である部分木は左からそれぞれ t[1]，t[2] で表される．また，t[1].root とは左の部分木の根である節点を意味するため，クラス B のメソッド b() の呼び出しを表す節点を指す．

我々は，繰り返し構造をループ構造の形式に圧縮する方法として R1 から R3 の 3 つのルールを考案し，

表 1 記法の定義

表記	意味
x,y	任意の節点
x.callNum	節点 x の子の数 (x.callNum $\geq$ 0)
x.method	x がもつメソッド
x.object	x がもつオブジェクト ID
t	任意の部分木
t.root	部分木 t の根である節点
t[i]	t.root の子である i 番目の部分木 ($1 \leq i \leq$ t.root.callNum)
L	部分木の列
L.size	部分木の列 L に含まれる部分木の個数
L[i]	部分木の列 L に含まれる i 番目の部分木 ($0 \leq i \leq$ L.size)

また，再帰構造の形式に圧縮する方法として R4 を考案した．

2.3.1 繰り返し圧縮ルール

本節では，繰り返し構造の圧縮を行う R1 から R3 までの 3 つのルールの説明を行う．まず，繰り返しの構造を圧縮する各ルールに共通する基本的な流れを説明し，次に，各ルールに固有の処理を順次説明する．

これらのルールに共通する流れは，実行履歴の木構造に対して，後順走査で各節点に以下の処理を行っていくことである．

> 節点 x について，その子である節点を根とする部分木の列から，「同一な構造」の部分木が繰り返されている部分を発見し，その繰り返し全体を表現するような部分木を作成し，繰り返し全体とそれを置き換える．次に，隣接した 2 つの部分木の列が繰り返されている部分を検出し，繰り返し全体を表現するような部分木の列に置き換える．これを，繰り返し 1 回当りの部分木の個数が x.callNum/2 より大きくなるまで繰り返していく．置き換えられた部分木や部分木の列には，繰り返しの回数も記憶しておく．

以上の流れを図で表したものが図 3 である．図中の

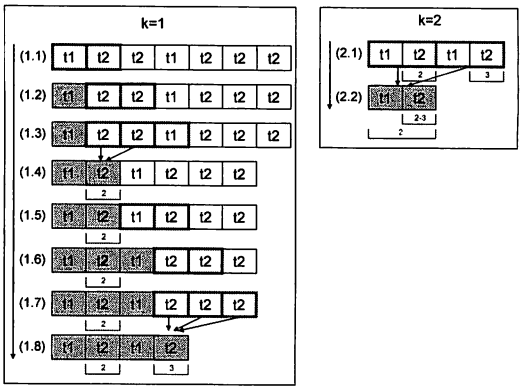

図 3 繰り返し構造の圧縮処理の流れ

t1, t2 は x の子を根とする部分木を表しており，同じ記号で表されているものは「同一な構造」と判定されるものであるとする．t1, t2, t2, t1... と並んでいるのは，x の子を根とする部分木がこのように並んでいるという意味である．k は繰り返し 1 回当りの部分木の個数を表す．太枠で囲まれている部分はその部分が繰り返しの比較対象になっていることを表している．k=1，つまり，1 個単位の繰り返しの検出から処理を進めていく．まず，左端の t1 とその次の t2 の比較が行われる (1.1)．これは同じものではないので比較対象を右へ 1 つずらす (1.2)．次の t2 同士は同一な構造であると判定され，その次の t1 は同一な構造ではないから (1.3) t2 が 2 個繰り返されているとしてここを 1 つ分に置き換える (1.4)．置き換え後，さらにその右側の t1 と t2 の比較を行う (1.5)．このような流れで処理を進めていき，右端の部分木までの比較を終えると，k=1 の時の処理を終える．次に，k=2 にして 2 個単位の比較を行っていく (2.1)．t1, t2 の 2 個単位の並びが繰り返されているので，この部分を 1 回分に置き換える (2.2)．右端まで比較を終えたので，k=2 の時の処理を終える．次に k=3 にすると，x.callNum = 2 であるから，k>x.callNum/2 が成り立つので，繰り返しを終え，節点 x についての処理を終了する．

R1 から R3 の各ルールは，上記の処理を以下の点で特殊化したものである．

- どのような部分木を「同一な構造」とみなすか
- 繰り返し全体の表現をどのように作成するか

- 繰り返しを検出した後，その次の比較に，繰り返し中のどの部分木列を用いるか

以下では，各圧縮ルールの処理の詳細を説明する．

R1:完全な繰り返し

圧縮ルール R1 では実行履歴中から，完全に同一な構造が繰り返されている箇所を検出し，圧縮する．つまり，繰り返し圧縮処理中の同一性の判定において，2 つの木の構造が等しく，かつ，それぞれの節点のメソッドとオブジェクトが等しいものを同一な構造と判定する．

同一性を判定する木構造を t_1，t_2 とし，R1 における木構造の同一性を $t_1 \equiv t_2$ と表現すると，$t_1 \equiv t_2$ とは，t_1.root.method = t_2.root.method かつ t_1.root.object = t_2.root.object であり，t_1.root.callNum = t_2.root.callNum，t_1[i] $\equiv$ t_2[i](∀i, 1 ≤ i ≤ t_1.root.callNum) を満たすことを表す．

また，同一性を判定する部分木列を L_1，L_2 とし，R1 における部分木列の同一性を $L_1 \equiv L_2$ と表現すると，$L_1 \equiv L_2$ とは，L_1.size = L_2.size であり，かつ L_1[i] $\equiv$ L_2[i](∀i, 1 ≤ i ≤ L_1.size) が成り立つことを表す．

また，繰り返し全体の表現として，繰り返し 1 回目の部分木列をそのまま用いることとする．図 3 の (1.3) や (1.7) のように繰り返しを検出した場合の次の部分木列との比較には，繰り返し中のいずれを用いてもよい (結果は変化しない)．このルールでは繰り返し回数を記録しておけば，元の実行履歴の内容を損なうことなく圧縮できる．

R2:オブジェクトが異なる繰り返し

R2 では実行履歴中から，オブジェクト ID のみが異なる構造が繰り返されている箇所を検出し，圧縮する．つまり，同一な構造であるかどうかの判定において，2 つの木の構造が等しく，かつ，それぞれの節点のメソッドが等しいものを同一な構造と判定する．

同一性を判定する木構造を t_1，t_2 とし，R2 における木構造の同一性を $t_1 = t_2$ と表現すると，$t_1 = t_2$ とは，t_1.root.method = t_2.root.method であり，t_1.root.callNum = t_2.root.callNum，t_1[i] = t_2[i](∀i, 1 ≤ i ≤ t_1.root.callNum) を満たすことを表す．

また，同一性を判定する部分木列を L_1，L_2 とし，R2 における部分木列の同一性を $L_1 = L_2$ と表現すると，$L_1 = L_2$ とは，L_1.size = L_2.size であり，かつ L_1[i] = L_2[i](∀i, 1 ≤ i ≤ L_1.size) が成り立つことを表す．

このルールではオブジェクト ID は比較しない．そして，繰り返し全体を表現する部分木列としては，R1 のように繰り返し 1 回目の部分木列をそのまま用いるのではなく，1 回目の部分木列の構造を基に，繰り返し中に出てくるオブジェクト ID を全て統合した部分木列を作成して用いる (図 4)．繰り返しを検出した場合に次の部分木列との比較には，R1 と同様，繰り返し中のいずれを用いてもよい．

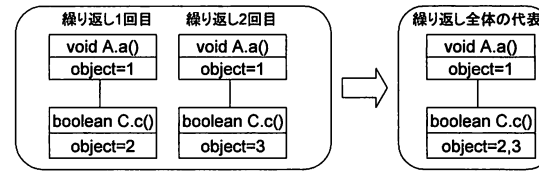

図 4　R2 における繰り返しの置き換え

このルールは R1 よりも多くの対象を圧縮できる．ただし，圧縮結果として表現される呼び出し構造は，同一クラスのオブジェクト群に対しての呼び出しを表すことになり，呼び出されたオブジェクトが特定できなくなるという点において，元の実行履歴全体を正確に表現しなくなる．

R3:欠損構造を含む繰り返し

R3 について説明をする前に，まず，メソッド呼び出し構造の包含関係について定義する．なお，以下の説明においては R2 と同様に，部分木や部分木列の比較には，各節点のメソッドのみを比較することとし，オブジェクト ID については考慮しないこととする．また以下では，t_1，t_2 は木構造，L_1，L_2 は部分木列を表す．

$t_1 \geq t_2$ とは，t_1.root.method=t_2.root.method かつ (t_1[1], t_1[2],... , t_1[t_1.root.callNum]) ≥ (t_2[1], t_2[2], ... , t_2[t_2.root.callNum]) である

ことを表す.

ここで，$L_1 \geq L_2$ とは L_2.size $= 0$ の場合，任意の L_1 に対して成立する．そうでない場合は，L_1.size $\geq L_2$.size であり，かつ p $= L_2$.size とすると L_1 から p 個の部分木 $L_1[i_1]$, $L_1[i_2]$, ... , $L_1[i_p]$, だけを順序は保存したまま ($i_1 < i_2 < ... < i_p$) 取り出したとき $L_1[i_x] \geq L_2[x](\forall x, 1 \leq x \leq p)$ が成り立つような添え字列 i_1, i_2,..,i_p が存在することを表す.

最後に，$t_1 > t_2$ とは $t_1 \geq t_2$ であり，かつ $t_1 \leq t_2$ ではないことである．$t_1 > t_2$ が成り立つことを，t_1 が t_2 を包含していると表現する．また，$t_1 \geq t_2$ の判定時にその部分木内の任意の階層の部分木列の比較において，p 個の要素として選ばれなかった部分木が表すメソッド呼び出し構造は，部分木 t_2 中では欠損しているという (図 5).

R3 では実行履歴中から，呼び出し構造の一部に欠損を含むような繰り返しを検出し，圧縮する．つまり，図 3 の同一性の判定において，繰り返しを検出していない場合は，比較対象である長さ k の 2 つの部分木列を L_1, L_2 とおいた時に，$L_1 \geq L_2$ または $L_1 \leq L_2$ である時同一であると判定する．繰り返しを検出した後は，検出済みの繰り返しの長さを r，繰り返しになっている部分木列を L_1, ..., L_r とし，次の比較対象の部分木列を L_{r+1} とすると，$L_i \geq L_j(\forall j, 1 \leq j \leq r)$ を満たす L_i について，$L_{r+1} \geq L_i$ または $L_{r+1} \leq L_i$ である時同一であるとする.

この繰り返し全体を表現する部分木列としては，繰り返し中の最大の要素，つまり，繰り返しの回数を r とすると，$L_i \geq L_j(\forall j, 1 \leq j \leq r)$ を満たす L_i を基にして，R2 の時と同様のオブジェクト ID の統合を行い，L_i 以外の要素で欠損している部分木に，欠損しているという情報を付加することで作成する.

このルールは，繰り返し毎に呼び出し構造が異なるような繰り返しをある程度圧縮することができるため，R2 よりも多くの対象を圧縮できる．しかし，繰り返し中の欠損部分の呼び出しは，メソッドが呼び出されたのか否かが不明になるという点において，元の実行履歴の構造を正確には表していないことになる.

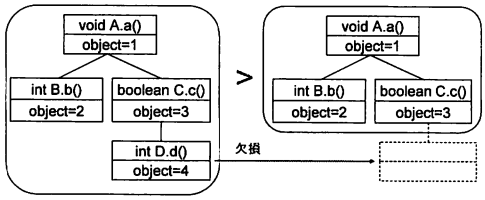

図 5 欠損構造の例

表 2 R4 で用いる記法の定義

表記	意味
x = y	x と y が同じ節点を指している
x.copy	x と同じ ID と同じメソッド名を持ち，子は持たない新たな節点
replace(x,y)	節点 x を親から切り離し，その場所に節点 y を繋げる
x.tree	x を根とする部分木
x.parent	x の親である節点
T	節点の列
T.size	節点の列 T に含まれる節点の個数
T[i]	節点の列 T に含まれる i 番目の節点 ($1 \leq i \leq$ T.size)

2.3.2 再帰構造の圧縮ルール

本節では，再帰構造の圧縮ルールである R4 について述べる．ルールの詳細に入る前に，R4 を説明するために，新たに表 2 に示す記法を定義する.

R4:再帰構造

R4 では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を検出し，圧縮する．ここではオブジェクト ID を考慮せず，同一メソッドであれば再帰として扱うものとする.

処理の方針は，まず，再帰構造になっているメソッドが呼び出されている部分で，木構造を分割していく．そして，分割された各部分木から，他の部分木全てについて，包含するかまたは同一な構造を持つような集合を選ぶ．ここでいう包含，同一な構造とは，R3 の説明で定義した関係を用いる．そして，その集合に含まれる部分木を組み

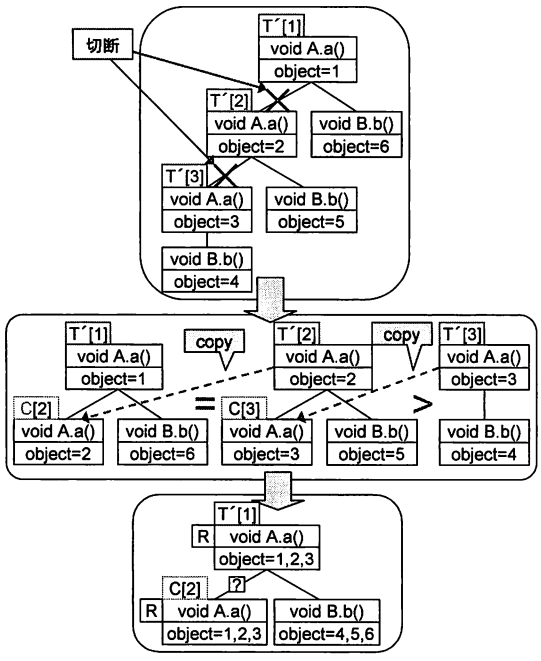

図 6　再帰構造の圧縮例

合わせて，木構造を再構成することで，再帰構造の簡潔な表現を作成する．

以下，再帰構造の圧縮を行う部分木の根である節点を x とし，圧縮処理の詳細を 4 段階に分けて説明する．なお説明を簡単にするために，再帰構造の圧縮処理の例を図 6 に示し，この図を参照しながら，説明を行う．

1. 再帰的に呼ばれているメソッドを表す節点の特定

メソッド呼び出しの順序に従って木構造を探索し，葉である節点 y を見つける．そして，x から y へ辿る経路上に存在する各節点の列 T を取る．T[1] は x，T[T.size] は y である．次に，T[t].method = T[r].method ($\exists$ r, $1 \leq t \leq$ T.size - 1, $t < r \leq$ T.size) を満たし，T[t] が再帰呼び出しとして圧縮処理を受けていないような，最小の t を見つける．t がなければ次の葉へ進む．t が存在すれば，T[t].method = T[r].method($t \leq r \leq$ T.size) を満たす T[r] を全て見つける (T[t] を含む)．その T[r] の列を $\acute{T}$ とする．つまり，$\acute{T}$ 中の各節点は，T 中に含まれる節点のうち，最初に現れる再帰的に呼ばれるメソッドへの，メソッド呼び出しを表す節点である．$\acute{T}$ の各要素は，対応する T 中の要素と同じ節点を指しているものとする．例えば，T=$\{x, a_1, b_1, a_2, b_2, y\}$ であり，a_1.method = a_2.method，b_1.method=b_2.method であるとすると，a_1.method と b_1.method が再帰的に呼び出されているが，t=2, r=4 で T[2].method = T[4].method が成り立つので，a_1 と a_2 が処理対象となる．このとき，$\acute{T} = \{a_1, a_2\}$ であり，$\acute{T}$[1]=T[2], $\acute{T}$[2]=T[4] となる．また，図 6 で言えば，左端の葉に到達した時にメソッド A.a() が 3 階層の再帰呼び出しになっているので，根からそのまま $\acute{T}$[1]=T[1]，$\acute{T}$[2] = T[2]，$\acute{T}$[3] = T[3] となる．

2. 再帰呼び出しの節点の切り離し (copy への置換)

次に，$1 \leq i \leq \acute{T}$.size を満たす全ての i について，$\acute{T}$[i].copy を作成し，これらの列を C とする．($\acute{T}$[1].copy = C[1])．そして，$\acute{T}$[i].parent が存在するなら，replace($\acute{T}$[i],C[i]) を行う．つまり，再帰呼び出しを表す節点 $\acute{T}$[i] をその親から切り離して，代替の節点 C[i](C[i].object = $\acute{T}$.object, C[i].method = $\acute{T}$[i].method) に置換する．ここで，$\acute{T}$[1] が x であり，実行履歴の木構造全体の根であった場合は parent が存在しない．その場合は，C[1] には置換しない．なお，この処理は図 6 で言えば，上段の状態から切断と copy を行った中段の状態へ移行する処理に相当する．

3. 代表元となり得る節点の選別

次に，R3 の説明で定義した包含関係を用いて，$\acute{T}$ の中から代表になる節点を選ぶ．ここで，部分木 A が部分木 B に包含されないことを，A ! < B と表す．また，部分木 A と B が等しいという関係には，R2 で定義した関係 A=B を用いる．また，等しくないことを A ! = B と記述する．$\acute{T}$[i].tree ! < $\acute{T}$[j].tree ($\forall$j,$1 \leq j \leq \acute{T}$.size) かつ $\acute{T}$[i].tree != $\acute{T}$[j].tree($\forall$j,$1 \leq j <$ i) を満たす $\acute{T}$[i] を全て見つけ，その数を m 個 (m$\geq$1) とし，そ

れぞれ $\acute{T}[i_1]$, $\acute{T}[i_2]$, ... , $\acute{T}[i_m]$ とする．これら $\acute{T}[i_1]$, $\acute{T}[i_2]$, ... , $\acute{T}[i_m]$ を $\acute{T}$ の代表元と呼ぶ．このとき，$\acute{T}$ 中の節点以下の部分木は，いずれかの代表元以下の部分木に包含されるか等しいものになる．図6で言えば，中段左の $\acute{T}$[1].tree は，$\acute{T}$[1].tree ! < $\acute{T}$[2] であり，$\acute{T}$[1].tree ! < $\acute{T}$[3] であるため代表元の1つである．しかし，$\acute{T}$[2].tree = $\acute{T}$[1].tree, $\acute{T}$[3].tree < $\acute{T}$[2].tree であるため，中央の $\acute{T}$[2].tree と右の $\acute{T}$[3].tree は代表元にはならない．

また，この比較の際に，R3 の時と同様に，対応する各オブジェクトの ID や欠損構造の情報を追加していく．その後，$\acute{T}[i_1]$, $\acute{T}[i_2]$, ... , $\acute{T}[i_m]$ の各節点と，C 中の各節点 C[i_1+1], C[i_2+1], ... , C[i_m+1] について，今回，再帰呼び出しとして圧縮処理を受けた節点に，それらが再帰呼び出しとして圧縮されたことを記録し，それら全てのオブジェクト ID を統合する．例えば，図6では，代表元が $\acute{T}$[1] だけなので m=1 である．よって，$\acute{T}$[1] と C[2] に，再帰呼び出しの圧縮処理を受けた記録をつける．図中ではこれを節点に “R” を付けて表している．また，各節点はそれぞれ対応する節点のオブジェクト ID が異なっている．さらに，$\acute{T}$[3] では A.a() の呼び出しが欠損しているため，オブジェクト ID の統合と，欠損構造の情報をつける．図中では欠損構造は辺に “?” を付けて表現している．

4. 代表元による木構造の構築 (copy との再置換)

まず，C[1].parent があれば，replace(C[1],$\acute{T}[i_1]$) を行う．つまり，節点 $\acute{T}$[1] の親から再帰構造全体が切り離されているなら，最初の代表元をそこに繋ぐ．ここで $\acute{T}$[1] が x であり，かつ実行履歴の木構造全体の根である場合は，C[1].parent が存在しない．この場合は，木構造全体を $\acute{T}[i_1]$.tree と置き換える．これは，x が実行履歴の木構造全体の根であり，再帰呼び出しの一部だった場合は，最初の代表元を新たな根とするということである．次に，m ≥ 2 ならば，全ての 2 ≤ j ≤ m について，j の小さい方から順に，replace(C[$i_{j-1}+1$],$\acute{T}[i_j]$) を行う．つまり，j-1 番目の代表元以下の部分木から，葉としてついている C[$i_{j-1}+1$] を切り離して，次の代表元をそこに繋いでいく．最後に，もし，葉 y が $\acute{T}[i_m]$.tree に含まれていれば，y についてもう一度 1 からの処理を行う．含まれていなければ $\acute{T}[i_m]$.tree 中に含まれる葉である，C[i_m+1] について 1 からの処理を行う．これは，再帰構造の圧縮を受けた T 中の葉である y か，あるいは y を含む部分木が切り離された結果，T 中の節点の1つと置き換えられて新たに葉となった C[i_m+1] のどちらかについてもう一度同じ処理をするということである．これによって，圧縮による木構造の組み換えが起こっても，木構造の順序的にその葉より前にある葉には既に到達済みであり，後にある葉には到達していないという状態を保つことができる．なお，図6のケースでは，まず，唯一の代表元である $\acute{T}$[1] を新たな根とする．そして，m=1 であり，i_m=1 であり，葉 y が残っていない為，新たに葉となった C[2] について，1 から処理を行う．それ以降は，圧縮済みの再帰呼び出ししか検出されずに，図6の圧縮処理は終了する．

このルールは圧縮効果そのものよりも，再帰的な呼び出し回数の深さの差を緩和することで，繰り返し圧縮ルールの効果を高めることを目的としている．

2.4 シーケンス図生成への適用

2.3 節で述べた手法で圧縮された実行履歴を基に，シーケンス図の作成を行う．繰り返し回数等の，圧縮結果を基にした情報を注釈として表現することで，より分かりやすい図を作成する．

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表現するかを述べる．

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7)．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．

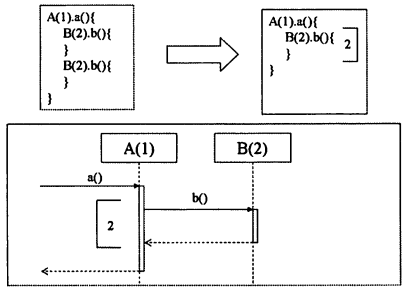

図 7　R1 適用部から作成される図

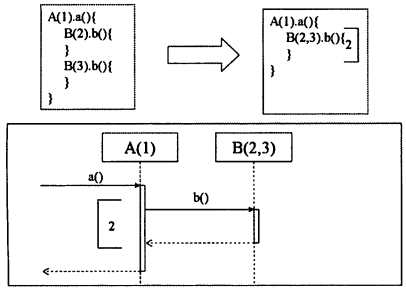

図 8　R2 適用部から作成される図

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，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 8)．

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箇所については，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 通りを描画する (図 9)．

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 10)．

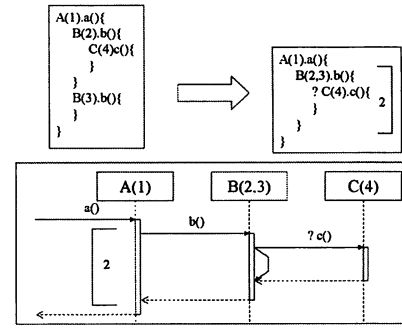

図 9　R3 適用部から作成される図

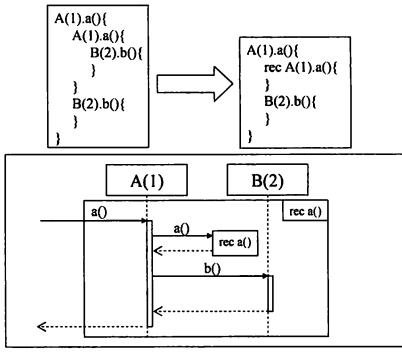

図 10　R4 適用部から作成される図

3　適用実験

表 3 に示した 4 つの Java プログラムと解析対象機能に対して，本手法の適用実験を行った．

実験ではまず，各プログラムの各機能を実行し，メソッド呼び出しの実行履歴を取得した．実行履歴の取得には Java Virtual Machine Profiler Interface(JVMPI) を利用して実装した実行履歴取得ツールを用いた．利用した JVM のバージョンは 1.5.0 である．ここで，java，javax，org，com，sun パッケージ及びそれらのサブパッケージ以下のクラスのメソッド呼び出しはライブラリとみなして実行履歴として取得しなかった．これは，Java のライブラリ部分の

表 3 実験対象プログラム

プログラム名	説明	解析対象機能
jEdit	テキストエディタ	テキストファイルの読み込み，表示
Gemini	コードクローン分析ツール	コードクローンの検出，結果の表示
scheduler	スケジュール管理ツール	スケジュール記述
LogCompactor	本ツールの実行履歴圧縮部	実行履歴の読み込み．圧縮ルール R2 の適用

表 4 圧縮結果

	圧縮前	R4 後	R1 後	R2 後	R3 後	圧縮率 (%)
jEdit	192005	185517	139407	37291	27963	14.56
Gemini	208360	205483	57365	1954	1762	0.85
scheduler	4398	4398	3995	238	147	3.34
LogCompactor	11994	8874	8426	208	105	0.88

動作を実行履歴上から除くことで，ユーザプログラム内の動作に限定した図を作成できるようにするためである．ただし，jEdit はそれ自身のクラス群が org.gjt.sp.jedit パッケージ以下に置かれているため，このパッケージについてはフィルタリングを行わなかった．

次に，考案した 4 つの圧縮ルールを R4，R1，R2，R3 の順に実行履歴に適用し，圧縮効果を測定した．この適用順序をとった理由は，R4 の説明で前述したとおり，先に再帰構造の圧縮を行い，再帰構造の階層の深さの差をなくすことで，後に適用する繰り返し圧縮ルールの効果が高くなることが期待できるからである．これによって，最終的な圧縮効果が最も高くなる．また，R1，R2，R3 については，R1 で圧縮可能な部分は R2 と R3 でも圧縮可能であり，R2 で圧縮可能な部分は R3 でも圧縮可能である．そのため，最終的な圧縮効果は R3 のみを適用した場合と同じであるが，それぞれの効果を測定するために，この順番で適用した．

最後に，圧縮結果からシーケンス図の生成を行った．また，scheduler について，設計時に作成されたシーケンス図と生成した図との比較実験も行った．

3.1 圧縮結果

それぞれの実行履歴の圧縮前と各ルール適用後のメソッド呼び出し回数を表 4 に示す．なお，表 4 中の圧縮率は次式で定義する．

$$\text{圧縮率} = \frac{\text{全ルール適用後のメソッド呼び出し回数}}{\text{圧縮前のメソッド呼び出し回数}} \times 100(\%)$$

この値が小さいほど圧縮効果が高いということになる．

まず，R4 では圧縮効果よりも再帰構造の再帰呼び出し回数の差を緩和することを目的としているため，圧縮効果はあまり高くなかった．しかし，再帰呼び出しを用いて実装されている部分が多い LogCompactor の実行履歴では，20%程度まで圧縮することができた．

次に R1 では，Gemini 以外の実行履歴には完全に一致する繰り返し部分が少ないためか，あまり圧縮効果は得られなかった．Gemini の実行履歴には単純な繰り返しが多かったため，高い効果を発揮し，25%程度まで圧縮した．

R2 は実験で用いた全ての実行履歴に対して高い圧縮効果を示しており，非常に有効であることがわかった．これは，オブジェクト指向言語で書かれたプログラムが，そのループ内において，同じオブジェクトへの処理を繰り返すのではなく，いくつかのオブジェクトの集合に対して，順に同じ処理を繰り返していく場合が多いことや，ループ内で毎回一時的に生成されるオブジェクトを利用すること等が原因だと考えられる．

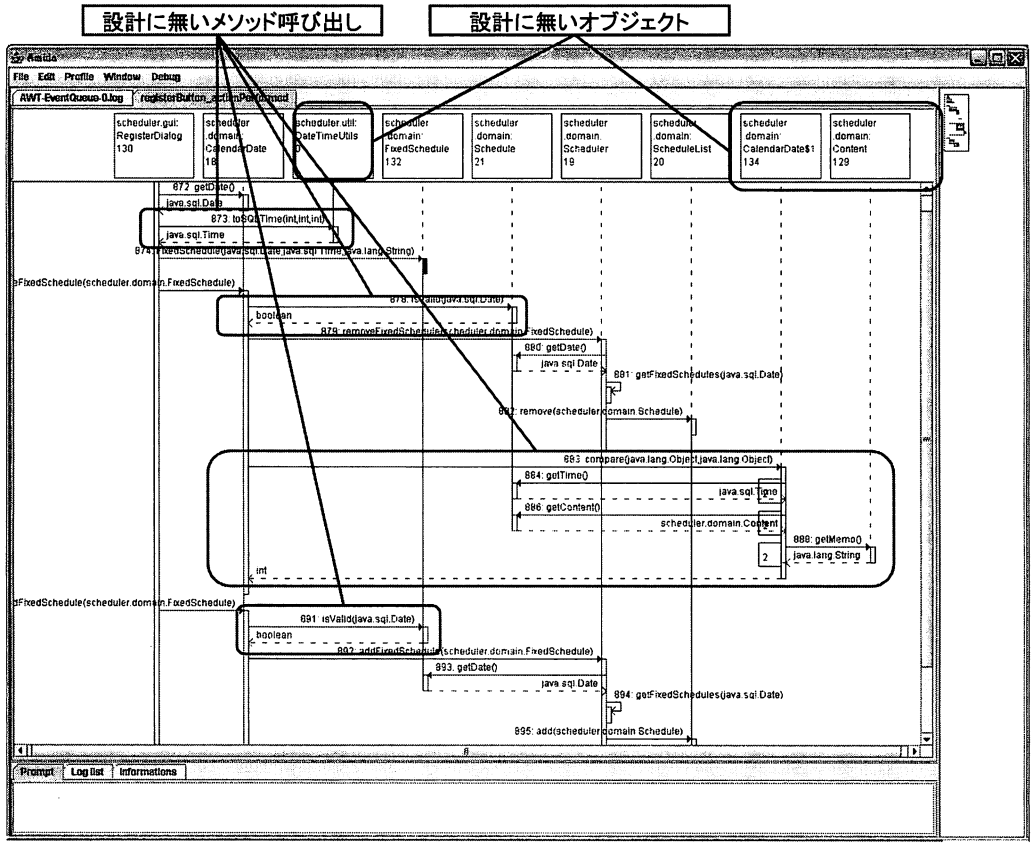

図 11　生成したシーケンス図 (scheduler)

R2 を適用した時点で，内部で分岐が発生しないループ構造は圧縮されてしまっている．さらに R3 を適用してみると，圧縮前のメソッド呼び出し回数が少ないものに対してはさらに 6 割程度までの圧縮が可能であったが，多いものにはあまり効果がなかった．この理由として，呼び出し回数が少ない実行履歴では，分岐が単純な欠損構造で表現できることが多いことに対して，呼び出し回数が多い実行履歴は，呼び出し階層が深く複雑な分岐構造になるため，R3 では効果が薄くなったと考えられる．

これら 4 つのプログラムに対する圧縮率は 0.85 ～ 14.56%となった．Gemini や Logcompactor の実行履歴の圧縮率は，それぞれ 0.85%，0.88%と良い結果が出ている．LogCompactor の実行履歴は，元のメソッド呼び出しが 11994 回であったものが，105 回まで圧縮されており，情報の圧縮が十分に行われていると考えられる．また，Gemini の実行履歴は最終的に 1762 回とやや多めだが，元の 208360 回から考えれば，情報として十分判読可能なサイズまで圧縮できている．scheduler，jEdit の実行履歴の圧縮率は 3.34%，14.56%であった．scheduler は圧縮後の実行履歴を見ると，メソッドの呼び出し回数が 147 回と十分少なく，また，全ての繰り返しについて圧縮が行われていたことが分かった．

しかし，jEdit の圧縮後の実行履歴には，繰り返し部分が多く圧縮されずに残っていた．これらは，プログラムのループ構造の中で実行時の条件によって複雑に分岐する構造になっているため，提案手法では圧縮できていなかった．このような構造には，静的解析から分岐構造をあらかじめ解析しておくことで，さらなる圧縮ができると考えている．

なお，最も圧縮処理に時間がかかったのは jEdit の実行履歴であった．この履歴に対して，CPU Pentium4 3.2GHz，メモリ 2GB の環境で実行時間は 2

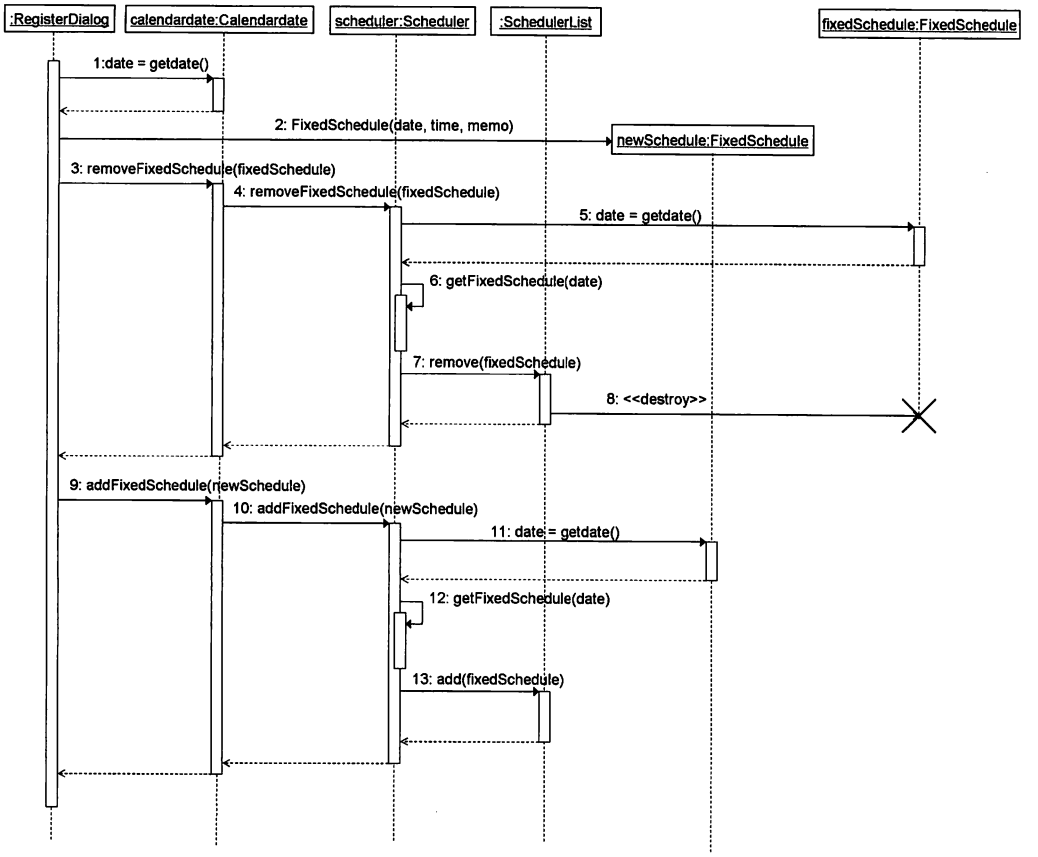

図 12 設計段階のシーケンス図 (scheduler)

分程度であった．本手法は，1 度圧縮処理を行ってしまえば，以降はその結果を使ってシーケンス図を作成すればよいので，この程度の時間であれば十分実用的であると考える．

3.2 シーケンス図の生成と設計時の図との比較

次に，圧縮結果からシーケンス図の作成を行った．図 11 は scheduler の圧縮後の実行履歴から生成されたシーケンス図である．図中には圧縮された繰り返しが表示されている．

また，作成された図が対象プログラムの実行時の動作を正しく反映しているかを調べるために，設計時に作成されたシーケンス図 (図 12) と，ツールから生成したシーケンス中の該当部分の図との比較を行い，それらの間の相違について調査を行った．対象としたプログラムはスケジュール管理ツール scheduler であり，比較したシーケンス図中に記述されているのは，スケジュールの削除を行う処理である．

両者のメソッド呼び出しの状況を比較した結果，全体の処理の流れとしては，ほぼ同様の構造が描かれていることが分かった．しかし，その中で，設計時にはないメソッド呼び出しやオブジェクトが，実行履歴から生成した図には存在した．図 11 中で囲まれている部分が該当部分である．

これらについて，該当処理のソースコードを確認したところ，スケジュールの削除処理の直後に，スケジュールデータのキャッシュを削除する処理が実装時に追加されていることが分かった．そして，生成した図中にのみ現れたメソッド呼び出しやオブジェクト群は，このキャッシュの削除処理に関連するものであった．また，他の部分のソースコードも確認したところ，このプログラムは全てのスケジュールデータが，ある 1 つのインスタンスで一元的に管理される構造になっていた．しかし，各日付を表すインスタンス毎

に，それらの日付に関連するスケジュールデータをキャッシュしており，個別の日付に関するスケジュールの取得処理などを高速化していることが分かった．この例は，(設計時に考慮されていなかったなどの何らかの理由により) 設計時の図には存在しない処理が，ソースコードで実装されている事例であり，保守作業などを行う際には知っていなければならない内容である．このように，本手法によって生成したシーケンス図は設計時の図より，実際のプログラムの動作を正しく表現している．

本手法では，実際のプログラムの実行時のオブジェクトの動作を図示し，動作理解に役立てることが目的であるため，このように実装時に追加された処理が図中に出現することは望ましい結果であり，本手法から作成する図がより正確にプログラムの実行時動作を表していることが確認できた．また，生成したシーケンス図は各ルールによって圧縮され，簡潔な図になっていたため，設計時の図と同様の処理を行っている箇所を容易に特定することができた．

4 考察

4.1 他のプログラミング言語への適用可能性

本手法で使用する実行履歴は，個々のメソッド呼び出しについて，呼び出しを受けたオブジェクトのオブジェクト ID と呼び出されたメソッドのメソッド名，引数の型，返り値の型，そのメソッドを実装しているクラス名，メソッド呼び出しの終了記号を記録しているものである．これらの情報はオブジェクト指向言語の一般的な概念のみを利用しているため，本手法はオブジェクト指向言語全般に適用可能である．ただし，それぞれの言語や実行系に合わせた実行履歴取得システムを用意する必要がある．

我々は現在までに，Java 言語に対する実行履歴取得ツールを作成しており，本稿の適用実験で用いている実行履歴はこれによって取得したものである．このツールは Java Virtual Machine Profiler Interface を用いて実装している．このインタフェースを用いたプロファイラを JavaVM の実行時にオプションとして指定することで，メソッド呼び出しなどの，実行時に発生する特定のイベントについての情報を VM から受け取ることができるようになっており，容易に実行履歴が取得できる．

現在のところ，他の言語の実行履歴取得ツールは実装していないが，ソースコード中にロギングコードを埋め込んで情報を出力させるようなツールを作成することで，同様の形式の実行履歴を取得することが可能であると考えられる．

4.2 マルチスレッドへの適応

現在の圧縮ルールは単一スレッドしか考慮していない．そのため，マルチスレッドプログラムでは，スレッド毎のシーケンス図を表示している．しかし，複数のスレッドが協調して動作するようなプログラムを理解するには，それらの相互の関連性を考慮した圧縮処理が必要になると考えられる．また UML のシーケンス図においても，単純な形態であれば複数のスレッドを同一図上に表現することは可能であるが，同じオブジェクトへのメソッド呼び出しが同時に発生した場合などは表現することができない．そのため，より複雑なマルチスレッドの動作を図示できるような，図の拡張が必要であると考える．このような，マルチスレッドプログラムへの手法の拡張は今後の課題である．

4.3 圧縮による情報の欠落

提案している 4 つの圧縮ルールでは，繰り返し構造や再帰構造全体をまとめて抽象化した形に置き換えることで，図を小さくしている．ここで，繰り返しや再帰に対して，呼び出しを受けるオブジェクト群のクラスや繰り返しの回数を概略情報とし，繰り返しや再帰一回ごとに異なる呼び出し回数や個別のオブジェクト ID を詳細情報と位置づける．その上で，ユーザーにまず概略情報を小さな図として示すことで，実行履歴を概略から詳細へと，トップダウンに理解することを目指している．

本手法で作成したツールでは，作成したシーケンス図中の特定の繰り返し部分を任意に展開し，部分的に圧縮される前の情報も対話的に閲覧できる．この機能を用いることで，圧縮された繰り返しや再帰構造の一回ごとの実行状態を参照したい場合は，その部分を

展開することで，より正確な情報を参照することができる．本手法は，1．繰り返しや再帰構造を圧縮することによって，それら一回毎の正確な情報をある程度犠牲にしつつ，全体の情報を表す表現に置き換えを行い，圧縮した小さな図を作成する．2．ツールのユーザは圧縮された状態の図を用いて，プログラムの実行時の動作の流れを見ていき，個別に注目したい部分があれば，任意に展開して一回ごとの正確な情報を参照する．このような流れで利用することで，できる限り必要な情報の欠落を起こすことなく，理解しやすい図を提供することができる．

4.4 圧縮アルゴリズムの設計評価

提案手法が，2.2節で述べた4つの項目を満たしているかを考察する．

1. シーケンス図上に，圧縮の適用結果が簡潔に図示できる必要がある

   提案した各ルールは，シーケンス図上に繰り返し構造，再帰構造，分岐構造を表す注釈を付加し，繰り返しによって同一の処理を受けるオブジェクトを統合することで，圧縮された状態をシーケンス図として表現することを可能にしている．通常のシーケンス図の記述様式では表現できてはいないが，シーケンス図の構造を崩すような形式ではないため，この程度であれば問題ないと考える．また，これらの構造はソースコード中でよく利用される構造であるため，ソフトウェア開発者にとっては身近な構造である．よって，本手法の利用者にはこれらの構造の理解は容易であると思われる．統合されたオブジェクト群が繰り返し中でそれぞれ同じ処理を受けることも，理解は難しくないと考える．なお，繰り返し構造と分岐構造はUML2.0では標準の形式を用いて記述可能であるため，今後はその形式に合わせていく予定である．

2. 圧縮の適用前の振る舞いの推測が容易である

   各ルールにおける圧縮結果は繰り返し構造，再帰構造，分岐構造と統合されたオブジェクトによって表現されている．R2，R3，R4の圧縮結果では，統合されたオブジェクトや分岐構造が登場する．これらは，4.3節で述べたように，統合されたオブジェクト群がどの順番でメソッド呼び出しを受けるか，分岐構造になっているメソッド呼び出しが何回目の繰り返しで発生するかという繰り返し1回毎の正確な情報が推測できなくなっている．しかし，R2，R3の圧縮結果として表示されるループ構造や，R4の圧縮結果として表示される再帰構造を見ることで，何回繰り返しが発生し，どのオブジェクト群が繰り返し中でメソッド呼び出しを受け，どのメソッドが繰り返し中で呼ばれるかといった，処理の流れの概要を推測することは可能である．適用実験においては，スケジュール管理ツールのschedulerの圧縮結果には，12回の繰り返しとして圧縮された処理があり，さらにその中で，28から31回の繰り返しが圧縮されている箇所があった．このことから，該当箇所では，外側の繰り返しで月ごとの処理を，内側の繰り返しで1日ごとの処理を行っている様子が容易に推測できた．このようなレベルの推測ができれば，十分実用的であると思われる．また，これらの構造は単純であり，特に理解が難しい構造ではないため，圧縮結果中の注釈表現の意味を知っていれば，誤って理解することは少ないと考えられる．

3. 特定の部分だけを局所的に選んで圧縮できる必要がある

   全ての圧縮ルールは木構造を圧縮対象としているため，任意の部分木に適用するか適用しないかを選択できるようになっており，局所的に圧縮，非圧縮を選択することができる．

4. 圧縮前の状態に戻せるようなアルゴリズムにする必要はない

   局所性が実現できているため，この項目については特に考察する必要は無い．

5 関連研究

シーケンス図とはオブジェクト間のメッセージ通信の様子を時系列に沿って記述することができる図である．しかし，静的解析からシーケンス図の生成を行う手法では，いくつかのオブジェクトのまとまりであ

る，クラス間でのメッセージ交換を記述することが，しばしば行われる．これは，静的解析から得られる情報からでは，動的に生成されるオブジェクトを追跡することが困難なためである．一部で，静的解析情報から可能な範囲で，動的に生成されるオブジェクトの動作を解析し，シーケンス図として表現する研究が行われている[12]．具体的には，オブジェクトの生成と参照の遷移を追跡し，ソースコード中の任意のメソッド呼び出し文について，呼び出し元になるオブジェクトと呼び出しを受けるオブジェクトを限定し，それら全てについてシーケンス図を生成するという作業を行っている．しかし，静的解析から判別できる動的な情報には限界があるため，実行時に動的に決定される要素については，完全には対応していない．

本手法と同じく，動的解析からシーケンス図を作成するという研究も多く行われている[1][4]．実行系列をそのままシーケンス図として表現するというソフトウェアも幾つかある[2][3]．また，デバッグ実行の途中でその実行時に集めた履歴を元にシーケンス図を表示し，デバッグ支援を行うツールもある[7]．しかし，動的解析から得られる情報は膨大な量に上るため，有用なシーケンス図を生成するためには，膨大な実行履歴の情報を削減する方法が必要である．これに対しては，実行履歴から動的スライスの計算を行い，指定されたスライス基準に関連するメッセージのみを抽出して表現する研究[4]や，アスペクト指向技術を用いて必要な情報のみを実行時に取得し，シーケンス図の作成を行っている研究[1]がある．また，Richnerらは，ユーザが呼び出し元と呼び出し先のオブジェクト，メソッド名までを指定することで，実行履歴中からその条件を満たすメソッド呼び出しを抽出し，シーケンス図を作成している[11]．その他にも，オブジェクトをクラスで分類し，動的な情報からクラス単位のシーケンス図を作成するツール[6]も存在する．これらの手法では，それぞれが想定する利用法において有用ではある．例えばデバッグ作業などで，不正な状態になっているオブジェクトに関連するメソッド呼び出しのみを抽出し，そのオブジェクトへのメソッド呼び出し履歴を図示することで，不正な状態になった原因を発見しやすくするなどの利用法がある．しかし，条件に合致する履歴のみを抜き出しているため，その条件に合致しない情報が図上から全て消えてしまう．このような図は，本手法が目指す，プログラム実行時の流れや，特定の機能に実現するためにどのようなオブジェクト群が動作するか，どのようなオブジェクト群が協調して動作するのかを理解する作業を支援する，という目的には不向きであると考える．

6 まとめ

オブジェクト指向プログラムの理解支援を目的として，動的解析情報から可読性の高いシーケンス図を作成する手法を提案し，実装を行った．実行時の動的解析から得られる情報は膨大な量に上るため，情報を圧縮し抽象化して表現し直すことが必要である．そこで，オブジェクト指向プログラムの構造を考慮した，4つの実行履歴圧縮ルールを考案した．また，提案手法を実現するためのツールを作成し，いくつかのJavaプログラムに対して適用実験を行った．その結果，実行履歴を大幅に圧縮し，簡潔なシーケンス図を作成することに成功した．

今後の課題としては，以下のようなものがある．

- 静的解析情報の併用．
  ソースコードの制御構造を利用して，より高度な圧縮を行う．
- マルチスレッドシーケンスの同一図中への表現．
  複数のスレッドを同一のシーケンス図上で表現することで，スレッド間の処理の関連性を図示し，バグの原因の特定などに役立てることが出来る．
- UML2.0[13]に準拠した図の作成を行う．

参考文献

[1] Gschwind, T. and Oberleitner, J.: Improving Dynamic Data Analysis with Aspect-Oriented Programming, in *European Conference on Software Maintenance and Reengineering*, 2003, pp. 259–268.

[2] Pauw, W. D., Jensen, E., Mitchell, N., Sevitsky, G., Vlissides, J. and Yang, J.: *Visualizing the Execution of Java Programs*, Lecture Notes in Computer Science, Volume 2269, 2002, pp. 151–162.

[3] IBM: Rational Test RealTime. http://www-306.ibm.com/software/awdtools/test/realtime/

[4] 小林隆志, 堅田敦也, 鹿内将志, 佐伯元司: プログラムスライシングを用いたJava実行系列からの部分シー

ケンス生成手法, ソフトウェア工学の基礎ワークショップ FOSE2004, ソフトウェア科学会, 2004, pp. 17–28.

[5] Lejter, M., Meyers, S. and Reiss, S. P.: Support for Maintaining Object-Oriented Programs, *IEEE Transaction Software Engineerings*, Vol. 18, No. 12 (1992), pp. 1045–1052.

[6] NASRA: j2u. http://www.nasra.fr/flash/NASRA.html

[7] Oechsle, R. and Schmitt, T.: *JAVAVIS:Automatic Program Visualization with Object and Sequence Diagrams Using the Java Debug Interface (JDI)*, Lecture Notes in Computer Science, Volume 2269, 2002, pp. 176–190.

[8] Pacione, M. J.: Software Visualisation for Object-Oriented Program Comprehension, in *International Conference on Software Engineering*, 2004, pp. 63–65.

[9] Pauw, W. D., Lorenz, D., Vlissides, J. and Wegman, M.: Execution Patterns in Object-Oriented Visualization, in *Conference on Object-oriented Technologies and Systems*, 1998, pp. 219–234.

[10] Reiss, S. P. and Renieris, M.: Encoding program executions, in *International Conference on Software Engineering*, 2001, pp. 221–230.

[11] Richner, T. and Ducasse, S.: Using Dynamic Information for the Iterative Recovery of Collaborations and Roles, in *International Conference on Software Maintenance*, 2002, pp. 34–43.

[12] Tonella, P. and Potrich, A.: Reverse Engineering of the Interaction Diagrams from C++ Code, in *International Conference on Software Maintenance*, 2003, pp. 159–168.

[13] Unified Modeling Language (UML) 2.0 specification nearing completion.

[14] Wilde, N. and Huitt, R.: Maintenance Support for Object-Oriented Programs, IEEE Transactions on Software Engineering, Vol. 18, No. 12 (1992), pp. 1038–1044.