Javaプログラムの実行履歴に基づくシーケンス図の作成

Extracting Sequence Diagram from Execution Trace of Java Program

谷口 考治*　石尾 隆†　神谷 年洋‡　楠本 真二§　井上 克郎¶

Summary. 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 is to compress redundant portions such as 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 extracts a sequence diagram from the result.

1　はじめに

オブジェクト指向プログラムでは，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が動作する．どのオブジェクトが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．そして，それは，保守過程において，プログラムの動作を理解する時も同じである．すなわち，プログラムの静的な記述であるソースコードを追いかけながら，システムによって生成されるオブジェクト群の動作を頭の中でイメージ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は容易に理解できるものではない [2] [5] [13]．そのため，オブジェクト指向プログラムの動作理解のために，プログラムから生成されるオブジェクト群の動的な振る舞いを解析し，理解支援を行う手法が求められている [7]．

そこで我々は，プログラムが実行時に生成するオブジェクト群の動作理解を支援するために，オブジェクト指向言語の1つであるJava言語のプログラムからUnified Modeling Language(UML) [11] のシーケンス図 (図 1) を作成する手法を提案する．本手法を用いることで，開発者は，設計ドキュメントが提供されていないプログラムのシーケンス図を作成してプログラムの動作を理解すること，設計段階で作成さ

*Koji Taniguchi, 大阪大学大学院情報科学研究科

†Takashi Ishio, 大阪大学大学院情報科学研究科

‡Toshihiro Kamiya, 科学技術振興機構さきがけ

§Shinji Kusumoto, 大阪大学大学院情報科学研究科

¶Katuro Inoue, 大阪大学大学院情報科学研究科

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ができる．また，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も想定している．

実行時のオブジェクト間の動作を解析するために，我々はプログラムの実行履歴を元に図の生成を行う．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 [8]．シーケンス図を作るために必要なメソッド呼び出しに関する情報だけに絞っても，その情報量は多い．そのため，できる限り情報量を削減しなければならない．そこで，我々は，実行履歴中の繰り返し構造に着目する．すなわち，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，これらを抽象化して簡潔な表現に置き換えることで，全体の情報量を削減し，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．

手法の具体的な内容としては，まず，解析対象とするプログラムを実行し，メソッド呼び出しの実行履歴を取得する．次に，この実行履歴には膨大な量のメソッド呼び出しが含まれているため，その中に含まれる繰り返しなどの特定のパターンを検出，圧縮し，抽象化した表現に置き換える．最後に，その結果を元にシーケンス図を作成することで，オブジェクト間のメッセージ通信を簡潔に利用者に提示する．また，抽象化して置き換えられた部分は，任意に圧縮前の情報を取り出し，詳細なシーケンス図も参照できるようにする．動的解析からシーケンス図を作成することの利点として，ソースコード中でループとして記述されていないが実際には同じ動作を繰り返す部分を，ループであると判定できることがあげられる．たとえば、プログラムの実行の最初で，異なったオブジェクトに同じ初期化を行っている部分はループとして検出される．また，実際にループを実行した回数もシーケンス図中に表示することができる．

我々は，オブジェクト指向言語，特に Java で実装された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．これらを用いて，繰り返しなどの特定のパターンを実行履歴中から検出，圧縮し，抽象化した表現に置き換えることで，情報量を大幅に減らすことができる．次に，各ルールによる圧縮が行われた部分をシーケンス図中に表現するための形式を考案した．これらを組み合わせることで，実行履歴中に含まれる膨大な量のメソッド呼び出しから，可読性の高いシーケンス図を作成する．本手法を実装した GUI ベースのシーケンス分析ツールでは，適用する圧縮ルールを選択したり，圧縮ルールが適用された個々の部分を展開することで詳細な情報を表示する機能を持っている．複数のプログラムに対する適用実験を通して，提案手法の有効性を確認した．

以降，2 節では本手法の詳細について，3 節では実装したシステムを用いた適用実験の結果とその評価について述べる．4 節では関連研究について説明し，5 節でまとめと今後の課題について述べる．

2　提案手法

2.1　概要

本手法ではまず，プログラムの動的解析を行い，得られた実行時情報を圧縮，抽象化していく．この結果を用いて，実行時の情報を反映した，プログラムの全体像

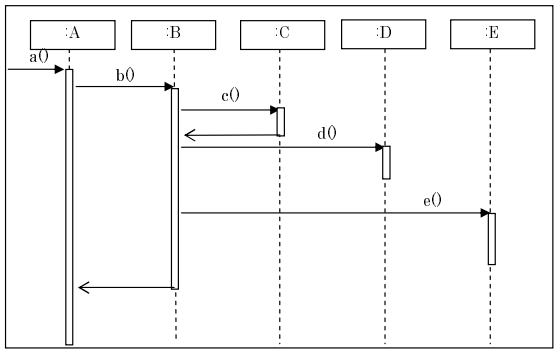

図 1　シーケンス図

を表す図を作成する．その全体図の中から，任意の圧縮された部分を展開することで，実行時の詳細な情報も参照できる．このような方法で，プログラム実行時の動作についての理解を支援する．

静的な解析方法と比較して，本手法のような動的な解析には以下のような特徴がある．

1. 解析対象として実行できるプログラムが必要である．
2. ソースコードが無い部分も解析が可能である．
3. 実行時の入力値に依存した動作を解析する．

このことから，本手法は保守過程におけるプログラム理解やドキュメントの再作成，デバッグ作業等において有効であると考えられる．

2.2　シーケンス図作成手順

本手法では次の Step1~Step4 を経てシーケンス図を作成する．

Step1: 解析対象プログラムへの入力決定
　解析対象となるプログラムへの入力を決定する．

Step2: 動的解析
　Step1 で決定した入力を元にプログラムを動作させ，動的解析を行い，メソッド呼び出しの実行履歴を取得する．

Step3: 実行履歴の圧縮
　Step2 で取得した実行履歴のメソッド呼び出し構造を解析，圧縮し，シーケンス図として表現できるサイズに加工する．

Step4: シーケンス図作成
　Step3 の結果を元に，圧縮情報等を反映したシーケンス図を作成する．

以降，2.3 節では Step2，Step3，2.4 節では Step4 の詳細について述べる．

2.3　実行履歴の圧縮

プログラムの全体像を簡潔に表す図を作成するために，実行履歴の圧縮処理を行う．本節では，実行履歴の形式と取得方法について説明し，圧縮手法の詳細について述べる．

```
Gemini(0).main(java/lang/String){
    GeneralManager(44753296).GeneralManager(java/lang/String){
        MDI(44736792).MDI(java/lang/String,GeneralManager){
            MDI(49860968).initComponents(){
                MDI(49860968).initMenuBar(){
                    MDI$2(44662040).MDI$2(MDI){
                    }
                    MDI$3(44666320).MDI$3(MDI){
                    }
                    MDI$4(44682200).MDI$4(MDI){
                    }
                }
            ...
```

図 2　取得する実行履歴の例

2.3.1　動的解析による実行履歴の取得

まず，プログラムを動的解析してその実行履歴を取得する．シーケンス図はメッセージ通信を表現する図であるため，ここでは，実行履歴としてオブジェクト間のメソッド呼び出しを記録する．具体的には個々のメソッド呼び出しについて，メソッド開始時にクラス名，オブジェクト ID，メソッド名，引数のシグネチャを記録し，終了時にメソッド終了記号を記録する．引数のシグネチャを取得するのは，メソッドのオーバーロード時にどのメソッドが呼ばれたかを特定するためであり，実行時に引数として与えられた値については取得しない．これらの情報を用いることで，実行時に呼び出されたオブジェクトとメソッドを特定し，呼び出し構造を再現することが可能となる．現在のところ Java プログラムを対象とした実行履歴取得システムを実装している．これには Java Virtual Machine Profiler Interface(JVMPI)[1] を利用した．

実行履歴取得システムは，対象とするプログラムの実行中に個々のメソッド呼び出しを捉え，実行履歴を保存するファイルに「クラス名 (オブジェクト ID). メソッド名 (引数)」，という形式で記録する．static メソッドの呼び出しについては，オブジェクト ID を 0 番としておく．

実際に取得した実行履歴の例を図2 に示す．この実行履歴では，プログラムが Gemini クラスの static として宣言された main メソッドから実行され，main メソッドの中で GeneralManager クラスの 44753296 番のオブジェクトに対して，GeneralManager メソッドが呼ばれている．これはコンストラクタであるため，ここで 44753296 番のオブジェクトが初期化されていることになる．さらにそのコンストラクタの中で，MDI クラスのコンストラクタが呼ばれ，という流れで実行されていく．途中に出現する閉じ括弧のみの行は，メソッドの終了を表している．

2.3.2　圧縮ルール

実行履歴中には，ループや再帰構造の中で発生するメソッド呼び出しの繰り返しが全て記録されている．後述する適用実験では，対象プログラムのいくつかの機能を実行しただけで，数十万回ものメソッド呼び出しが発生した．これらをそのままシーケンス図として表現すると，プログラム全体の動作を理解することが困難になる．そこで，実行履歴中からこれらの繰り返しを検出，圧縮し，抽象化した表現に置き換えることで，実行履歴の全体像を理解しやすい図を提示できるようにする．

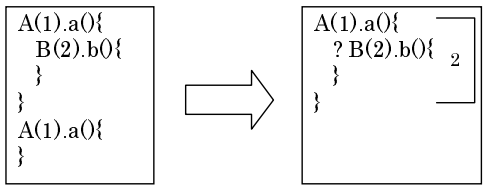

図 3　欠損構造の圧縮例

そこで，繰り返し構造と再帰構造を検出，圧縮する方法として，以下に示す R1 から R4 までの 4 つのルールを考案した．以下それぞれについて説明する．

R1:完全な繰り返し

実行履歴中から，完全に同一な呼び出し構造が繰り返されている箇所を検出し，圧縮する．完全に同一な呼び出し構造とは，その呼び出し構造内の各メソッド呼び出しが，同じオブジェクトに対する同じメソッドの呼び出しであることをいう．本手法では，実行時の引数の値については考慮しないため，異なる引数で何度も同じメソッドが呼び出された場合でも，引数の影響によってメソッド呼び出し関係が変化しない限りはこのルールで圧縮される．このルールでは繰り返し回数を記録しておけば，元の実行系列の内容を損なうことなく圧縮できる．その一方で，完全に同一な呼び出し構造は実行履歴中に多数存在するものではないため，圧縮による情報量の削減効果は小さい可能性が高い．

R2:オブジェクトが異なる繰り返し

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．このルールは R1 よりも圧縮効果が大きい．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，元の実行系列全体を正確に表現していない．

R3:欠損構造を含む繰り返し

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．欠損を含むような繰り返しの圧縮例を図 3 に示す．この例では，1 回目の A クラスの a() メソッドの呼び出しの内部で B クラスの b() メソッドの呼び出しが発生しているが，2 回目の a() メソッドの呼び出しでは b() メソッドの呼び出しは発生していない．これを，b() メソッドが欠損しているとみなし，図の右側のように「実行される場合と実行されない場合がある」という表現 (メソッド呼び出し前に “?”) をつけて圧縮を行う．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 よりも圧縮効果が高くなる．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，元の実行系列の構造を正確には表していないことになる．

R4:再帰構造

実行履歴中の呼び出し構造において再帰的に呼び出されているメソッドを検出し，圧縮する．再帰構造の圧縮例を図 4 に示す．ここではオブジェクト ID を考

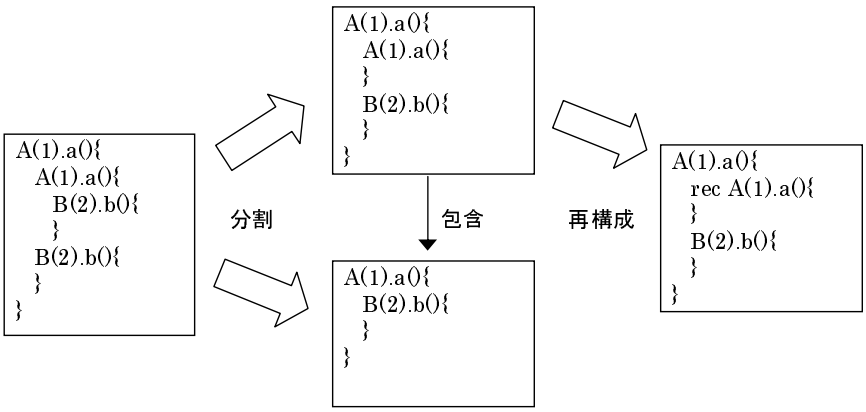

図 4　再帰構造の圧縮例

慮せず，同一クラスの同一メソッドであれば再帰として扱うものとする．具体的には，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，メソッド呼び出し構造を分割し，分割された各階層の中から，他の階層全てを包含するような集合を選ぶ．ここでいう包含とは，同一の呼び出し構造を持つか，R3 と同様の欠損構造を持つもののことを言う．そして，その集合に含まれる階層を組み合わせて，再帰構造の簡潔な表現を作成する．このとき，R3 と同様の欠損構造になっているものは,「実行される場合と実行されない場合がある」メソッドを含んでいるので，その部分には R3 と同様の表現を付ける．図 4 の例では，a() メソッドの呼び出しが再帰的に行われているので，この構造を a() メソッドの呼び出しの部分で分割する．その結果，分割された 2 つの階層は，図の中央上の部分が中央下の部分を包含しているので，中央上の部分のみを含む集合が全体を包含している集合となる．そして，この集合内の要素，つまり図の中央上の部分を用いて再帰構造の再構成を行っている．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　シーケンス図の生成

圧縮した実行履歴を元にシーケンス図の作成を行う．繰り返し回数等の，圧縮結果を元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．これらを表す形式は UML 1.5 [11] の仕様にはないが，UML2.0 [12] では表現可能になる予定であり，その形式に対応することは今後の課題である．

以下，実行履歴中で各圧縮ルールを適用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．

未圧縮部

圧縮ルールを適用しなかった場合や，圧縮ルールを用いても圧縮されなかった部分，圧縮結果を展開した部分については通常のシーケンス図と同様の形式で表現する (図 5)．

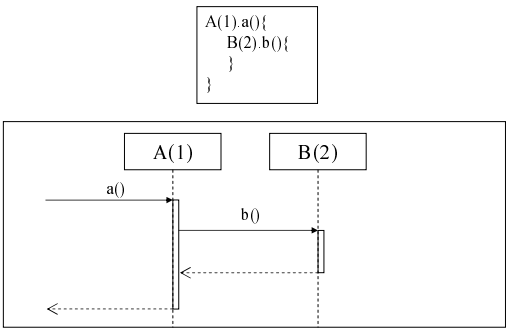

図 5　未圧縮部から作成される図

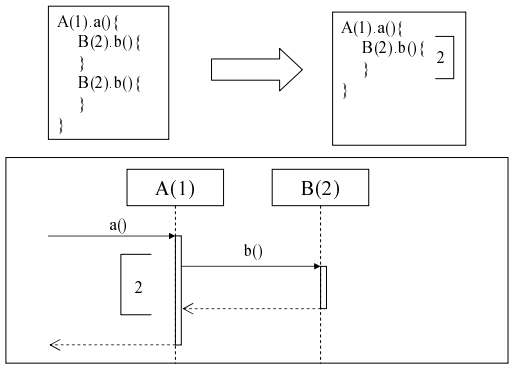

図 6　R1 適用部から作成される図

R1 適用部

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 6)．なお，このループ情報は以降の R2,R3 についても同様の形式で表現する．

R2 適用部

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 7)．この手法によって，同じ ID のオブジェクトが、シーケンス図上部に複数表れることがある．例えば，ある繰り返しの圧縮により，オブジェクト ID1 番と 2 番のオブジェクトが統合され，別の繰り返しの圧縮により，オブジェクト ID2 番と 3 番のオブジェクトが統合されたものが生成されたとする．このとき，2 番のオブジェクトは統合された 2 つのオブジェクト群に出現することになる．このようなことから，結果のシーケンス図は，繰り返しによって統合されるオブジェクトの組み合わせの数だけ，横に大きくなってしまう．この問題を改善するために，同じオブジェクトを共有する統合されたオブジェクトを，さらに 1 つのオブジェクト群に統合して表現する手法を取る．例えば，先ほどの例では，2 つのオブジェクト群を 1，2，3 番のオブジェクトを統合した 1 つのオブジェクトとして図中に表現する．この結果，ある 1 つのオブジェクトは，他のオブジェクトと統合されなかった場合の単体のオブジェクトと，他のオブジェクトと統合されたオブジェクト群の，高々2 つのオブジェクトで表現される．

R3 適用部

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，発生する場合としない場合がある呼び出しについては，それが呼ばれるシーケンスと呼ばれずに素通りするシーケンスの 2 通りを引く (図 8)．

R4 適用部

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

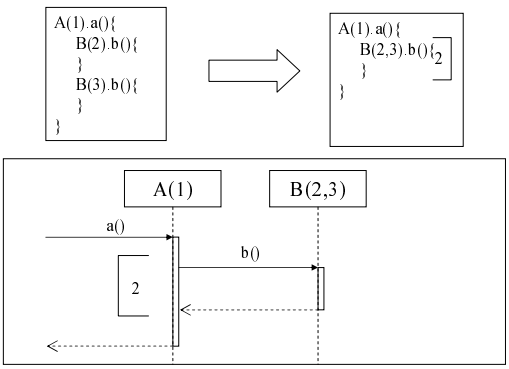

図 7　R2 適用部から作成される図

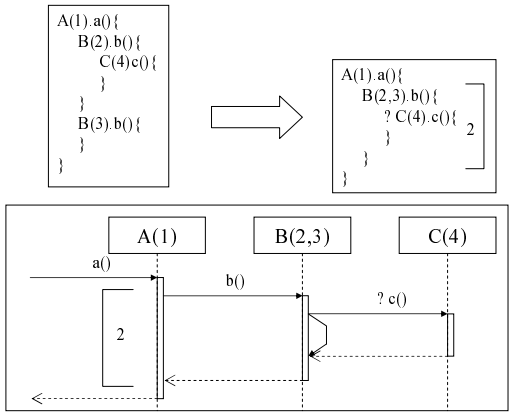

図 8　R3 適用部から作成される図

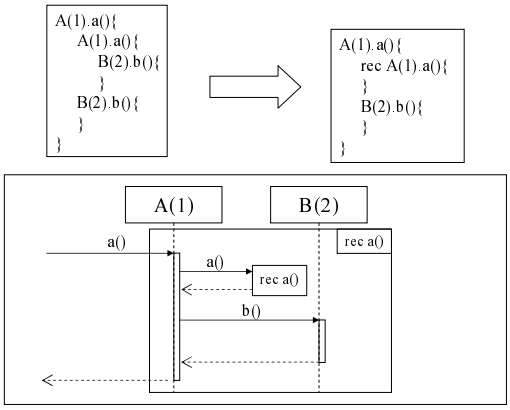

図 9　R4 適用部から作成される図

再起呼び出しを表す．そのため，再帰呼び出し構造全体を四角で囲み，内部で再起的に呼び出される部分では，外側の四角と同じ構造を表す四角へのシーケンスを引く (図 9)．

3　適用実験

提案した手法について実装を行い適用実験を行った．本節ではその内容及び結果と，結果に対する考察を述べる．

3.1　実験内容

実験には 4 つの Java プログラムに対して本手法を適用した．各プログラムの説明と解析対象とした機能は表 1 の通りである．

これらのプログラムの機能に対して動的解析を行い，実行履歴を取得した．

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，まず再帰構造の圧縮を行い，

表 1　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

その後，各繰り返し圧縮ルールを条件の厳しいものから適用することで，最終的な圧縮効果が最も高くなると判断したからである．

そして最後に，圧縮結果を元にシーケンス図の生成を行った．

3.2　実験結果と考察

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 2 に示す．なお，ここで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．本研究では，全体の大きさを十分に小さくすることで全体像を把握できるようにすることを第一の目標としている．そして，圧縮した後，ユーザが注目したい各部分については部分的に圧縮結果を展開しながら閲覧することを可能にすることで，圧縮率を高くしたことによる情報の損失の影響を小さくしている．

まず，R4 は圧縮効果よりも，再帰構造の再帰的な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，圧縮効果は高くない．それでも，LogCompactor のような再帰構造が多く現れる実行履歴では，多少の圧縮効果が確認された．

次に R1 では，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．しかし，単純な繰り返しが多い Gemini の実行履歴では高い効果を発揮した．

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場合があること等が原因だと考えられる．

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．これは，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，このルールでは効果が薄くなったと考えられる．

圧縮率は 0.85 ～ 7.22%となった．Gemini や Logcompactor の実行履歴の圧縮率

表 2　圧縮結果

	圧縮前	R4 後	R1 後	R2 後	R3 後	圧縮率 (%)
jEdit	228764	217351	178128	16889	16510	7.22
Gemini	208360	205483	57365	1954	1762	0.85
scheduler	4398	4398	3995	238	147	3.34
LogCompactor	11994	8874	8426	208	105	0.88

は 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は 3.34%，7.22%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 の実行履歴には，繰り返し部分が多く圧縮されずに残っており，元の呼び出し回数を考えても，この圧縮率では十分に情報量を削減できているとはいえない．このような実行履歴に対しては，さらなる圧縮ルールが必要である．

最後に，作成したシーケンス図の一例を図 10 に示す．ここでは scheduler の実行履歴から生成されたシーケンス図を例として挙げている．ループの繰り返し回数，R2 によって統合されたオブジェクト群へのシーケンス等が表示されている．

4　関連研究

シーケンス図とは本来オブジェクト間のメッセージ通信を記述する図である．しかし，静的解析からシーケンス図の生成を行う場合，単純な手法では，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しかできない．この問題に対して，静的解析情報から判別できる範囲でオブジェクトの生成とそれが参照される位置を追跡し，オブジェクト単位でのシーケンス図を作成する取り組みが行われている [9]．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．また，オブジェクト単位の記述にこだわらず，クラス単位のシーケンス図を作成し，さらにその中で，複数のクラスやメッセージを一つに統合するなどして，シーケンス図の抽象化を行っている例もある [10]．

また，動的解析からシーケンス図を生成するという手法では，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，実行履歴からスライスの計算を行い，指定された基準に関連するメソッド呼び出しのみを抽出して表現する研究が行われている [4]．具体的には，実行履歴中からスライス基準として指定したメソッド呼び出しに影響を与えうるメソッド呼び出しのみを図中に表現する．そして，この図を用いることで，スライス基準として選択したメソッド呼び出し内で発生する問題の，原因となりうる部分の動作のみを追跡することができる．また，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] もある．この研究では，指定した任意のクラスのオブジェクトが関連

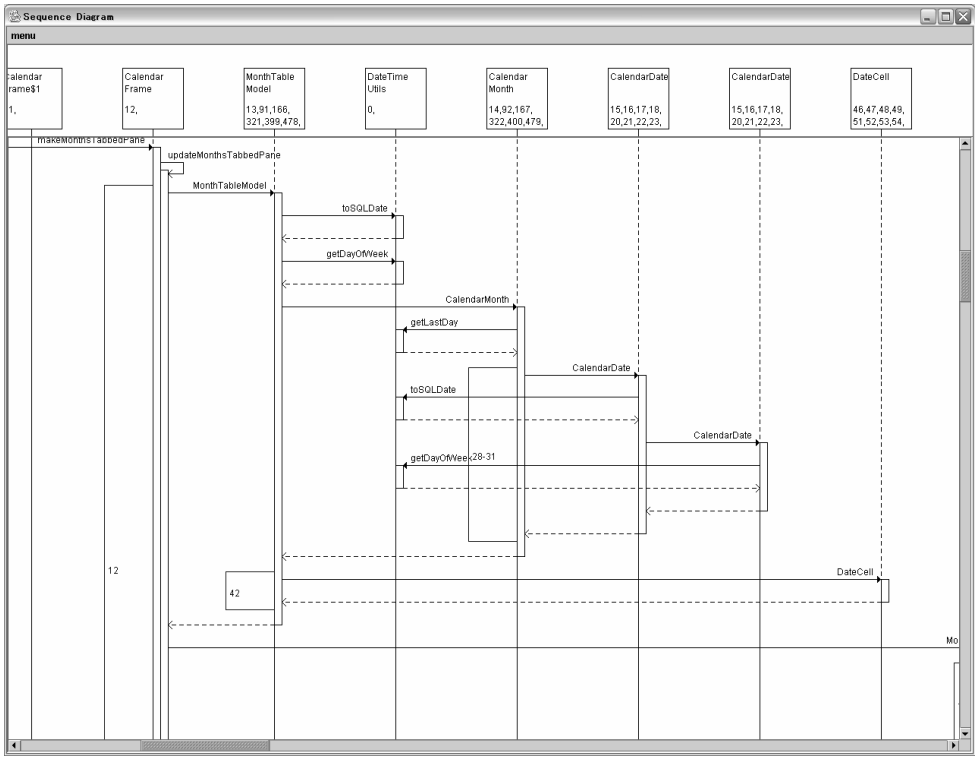

図 10　scheduler から生成されたシーケンス図

するメソッド呼び出しだけを実行履歴として取得している．そして，取得した実行履歴の中から特定のオブジェクトに関して，そのオブジェクトがメソッドが呼び出しを受けている部分，そのオブジェクトが他のオブジェクトのメソッドを呼び出している部分，などの条件を指定することで，指定したオブジェクトと他のオブジェクトの関連性をシーケンス図で表現することができる．また，オブジェクトをクラスで分類し，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が，本手法が目指す，実行時のオブジェクトの情報からプログラムの全体像を把握する，という目的には不向きであると考える．

5　まとめと今後の課題

オブジェクト指向プログラムの理解支援を目的として，動的解析情報からシーケンス図を作成する手法を提案した．

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，適用実験を行った．その結果，実行履歴の大幅な圧縮に成功した．さらに，圧縮結果をシーケンス図に示す表現を考案し，実際にシーケンス図の作成を行った．

今後の課題としては，さらに情報を抽象化して表現する為の圧縮ルールの考案や，

作成したシーケンス図に対しての評価などがあげられる．また，実行履歴を，それぞれが担当する機能ごとに分割することも考えている．1 つの実行履歴から複数の分割された図を作成できれば，情報量の多さという問題を，緩和できる．分割を行うためには，実行履歴の呼び出し構造をなんらかの機能的なまとまりに分類する手法と，その結果を図中に表現するアイデアが必要であり，現在検討中である．また，今後，UML2.0 [12] が正式に策定されれば，その形式に対応することも考えている．その他，ソースコードを静的解析した結果の情報を併用することや，マルチスレッドを利用するプログラムに対応することも今後検討していきたい．

参考文献

[1] Thomas Gschwind, Johann Oberleitner: Improving Dynamic Data Analysis with Aspect-Oriented Programming. Proccedings of the 7th European Conference on Software Maintenance and Reengineering, March 26–28, 2003.
[2] Dean F. Jerding, John T. Stasko, Thomas Ball: Visualizing interactions in program executions. Proceedings of the 19th international conference on Software engineering, pp.360-370, 1997.
[3] Java Virtual Machine Profiler Interface. *http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jvmpi/jvmpi.html*
[4] 堅田敦也, 鹿内将志, 小林隆志, 佐伯元司: Java 実行系列からのシーケンス図生成ツール. 情報処理学会ソフトウェア工学研究会, ウィンターワークショップ イン 石垣島, January 2004.
[5] Moises Lejter, Scott Meyers, Steven P. Reiss: Support for Maintaining Object-Oriented Programs. IEEE Transaction Software Engineerings. Vol.18, No.12, pp.1045-1052, December 1992.
[6] NASRA: j2u. *http://www.nasra.fr/flash/NASRA.html*
[7] Michael J. Pacione: Software Visualisation for Object-Oriented Program Comprehension. Proceedings of the 26th International Conference on Software Engineering, pp.63-65, 2004.
[8] Steven P. Reiss, Manos Renieris: Encoding program executions. Proceedings of the 23rd International Conference on Software Engineering, pp.221-230, 2001.
[9] Paolo Tonella, Alessandra Potrich: Reverse Engineering of the Interaction Diagrams from C++ Code. Proceedings of International Conference on Software Maintenance, pp.159-168 2003.
[10] 鳥井邦貴:Java コードから UML/シーケンス図へのリバースエンジニアリング. 信州大学大学院工学系研究科情報工学専攻, 修士学位論文,2003.
[11] Unified Modeling Language (UML) 1.5 specification. OMG, March 2003.
[12] Unified Modeling Language (UML) 2.0 specification nearing completion.
[13] Norman Wilde, Ross Huitt: Maintenance Support for Object-Oriented Programs. IEEE Transactions on Software Engineering, Vol.18, No.12, pp.1038-1044, December 1992.