修士学位論文

題目

シーケンス図生成のためのプログラム実行履歴の圧縮手法

指導教官
井上克郎 教授

報告者
谷口 考治

平成 17 年 2 月 14 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 16 年度 修士学位論文

シーケンス図生成のためのプログラム実行履歴の圧縮手法

谷口 考治

内容梗概

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群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．そのため，システムの動作を理解するためには，生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，また，1 つの機能を実現するために多くのオブジェクトが関連して動作をすることから，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．

そこで，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に対する動的解析結果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ことが多く，結果として作成されるシーケンス図が大きくなるという問題が生ずる．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．具体的には，4 つの実行履歴圧縮ルールとそれぞれの圧縮結果をシーケンス図に反映する表現方法を考案した．更に，提案手法を実装したツールを用いて適用実験を行い，圧縮ルールの有効性を確認した．

主な用語

リバースエンジニアリング
シーケンス図
プログラム理解
動的解析

目 次

1 まえがき **4**

2 シーケンス図とプログラム理解 **6**
2.1 UML シーケンス図 6
2.2 シーケンス図によるプログラム理解支援 7
2.3 リバースエンジニアリングによる図の作成 7

3 実行履歴圧縮手法 **9**
3.1 動的解析による実行履歴の取得 9
3.2 圧縮ルール 11
3.2.1 繰り返し圧縮ルール 12
3.2.2 再帰構造の圧縮ルール 20
3.3 シーケンス図生成への適用 24

4 ツールの実装 **28**
4.1 ツールの実装 28
4.2 ツールの構成 28
4.2.1 GUI 28
4.2.2 プロファイラ DLL 30
4.2.3 プロファイル部 31
4.2.4 コールツリー生成部 31
4.2.5 シーケンス図生成部 32
4.2.6 圧縮部 33
4.3 ツールの利用手順 33

5 適用実験 **37**
5.1 実験内容 37
5.2 圧縮結果 37
5.3 シーケンス図の生成 41

6 関連研究 **44**

7 むすび **46**

謝辞 **47**

参考文献 48

1 まえがき

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 [7][10][16][28]．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 [13][14][27]．

オブジェクトの動作を表現する方法として，UML[24] のシーケンス図 (図 1) がある．シーケンス図とは，オブジェクト間のメッセージ通信を，時間の流れに沿って表現できる図であり，この図を作成することで，あるオブジェクトがどのような順序で，どのオブジェクトとメッセージ通信を行うかを表現することができる．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，設計時に書か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表現する図の作成を行う．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は，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えばよい．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 [14][15][17]．シーケンス図を作るために必要なメッセージ通信である，メソッド呼び出しやオブジェクトの生成に関する情報だけに絞っても，その情報量は多く，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 [9]．そのため，できる限り情報量を削減しなければならない [14][17]．

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．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．

本手法を用いてシーケンス図を生成する手順は次のようになる．まず，解析対象とするプログラムを実行し，オブジェクトの生成とメソッド呼び出しに関する実行履歴を取得する．次に，この実行履歴には膨大な量のオブジェクトの生成とメソッド呼び出しが含まれているため，提案する手法により，その中に含まれる繰り返しなどを圧縮する．最後に，その結果を基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．なお，圧縮による情報の欠落の影響を小さくするために，圧縮された部分は，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．

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．各ルールは，それぞれ，繰り返しや再起構造のパターンを検出し圧縮を行う．そして，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，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情報から，簡潔な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図生成ツールでは，提案する圧縮ルールの中から適用するものを選択したり，圧縮ルールが適用された個々の部分を展開することで，元の詳細な情報を表示する機能を持っている．このツールを用いて，複数のプログラムに対する適用実験を行い，提案手法の有効性を確認した．

以降，2 章ではシーケンス図の詳細とプログラムから図を作成する方法について，3 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，4 章では提案手法を実装したツールについて説明し，5 章では適用実験の結果を述べその考察を行う．6 章では関連研究について，7 章でまとめと今後の課題について述べる．

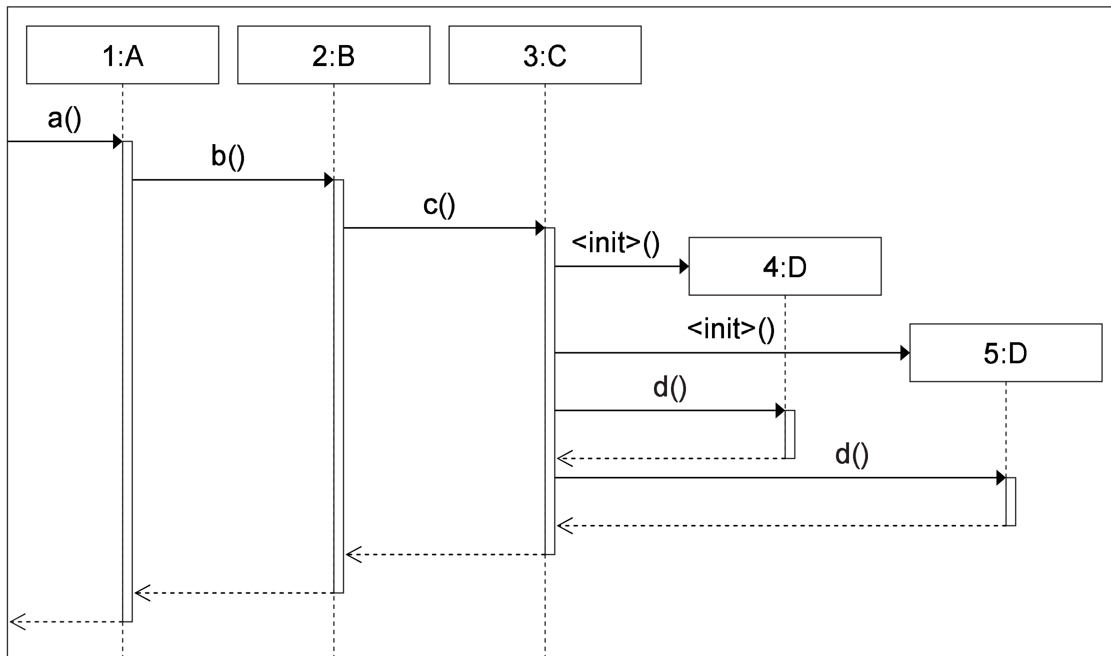

図 1: シーケンス図

2　シーケンス図とプログラム理解

2.1　UML シーケンス図

シーケンス図 (図 1) とは UML で定義されているインタラクション図の 1 つで，オブジェクト間のメソッド呼び出しやオブジェクトの生成などのメッセージ通信について，時系列に沿って示すことができる図である [24]．横軸はオブジェクトの種類を表しており，図 1 のように，図の上部には，図中に記述されるメッセージ通信に関連するオブジェクトが横方向に並べられる．縦軸は時間軸を表しており，下方に行くほど時間が経過していく．各オブジェクトの下部には縦方向に点線が引かれており，これがオブジェクトが生存する区間を示している．そして，個々のメッセージ通信について，時系列順に，送信元のオブジェクトから，送信先のオブジェクトに対して矢印を引く．送信されたメッセージがメソッド呼び出しだった場合は，メッセージを受けたオブジェクトは，そのメソッドの実行区間を縦長の長方形で表す．そしてメソッドが終了すると，呼び出し元のオブジェクトへ戻り辺の矢印を引く．メッセージがオブジェクトの生成だった場合は，生成されるオブジェクトをその高さに書く．このようにして，オブジェクト間のメッセージの様子を時系列に沿って表現する．

2.2 シーケンス図によるプログラム理解支援

シーケンス図は UML の図の 1 つであり，通常はシステムの設計時に作成される図である．この図を作成することで，設計者はシステムが実際に動作する際のオブジェクト間の関連を，設計時にイメージすることができ，また，他の設計者や開発者にオブジェクトの動作を分かりやすく示すことができる．

さらに，シーケンス図は，既存のプログラムを理解する作業においても有用である．というのも，オブジェクト指向プログラムにおいては，静的な記述であるソースコードから，プログラム実行時のオブジェクトの動作を理解することは難しく，ソースコードとは別にオブジェクトの動作を記述した文書が存在することが望ましいからである．その際に，シーケンス図を用いることで，各オブジェクトがどのような流れで，どのように関連してメッセージ通信を行うのか，を見ることができる．例えば，ソースコード中で操作しているオブジェクトは，どこで生成されて，どのタイミングでデータをセットされたのか，ということがシーケンス図を見ることによって理解できる．

2.3 リバースエンジニアリングによる図の作成

しかし，実際には設計時に作成されたシーケンス図は，オブジェクトやクラスの分け方が曖昧であったり，処理の流れが大まかに記述されていることがある．また，製作途中での仕様の変更があっても，新しい設計書が作成されないことや，実装段階においての問題のために，設計と異なる実装になることがあり，作成されたプログラムは設計時に作られたシーケンス図とは異なった動作をすることがある．また，そもそも，設計時にシーケンス図が作成されないことも有り得る．

そこで，作成されたプログラムを解析し，リバースエンジニアリングによりシーケンス図を作成することで，プログラムを理解する作業の支援やドキュメントの再作成を行うという手法があり，様々な研究が行われている [3][9][12][17][21][22]．詳細については 6 章で述べるが，これらの研究は大きく 2 つに分かれる．1 つは，静的解析から図の作成を行う研究であり，もう 1 つが，動的解析から行う研究である．静的解析から行う場合は，ソースコード全体の図が作成できる，解析がしやすい，作成途中のプログラムや，ソースコードの一部といった実行できないような単位の解析が行えるという利点がある．しかし，やはり静的な解析であるため，オブジェクトの動作を正確に把握することは困難である．一方の動的解析では，オブジェクトの動作を解析することが容易である，実際にプログラムが動作した部分のみを解析できるという利点があるが，プログラムが実行された部分しか解析できず，結果が入力値に依存してしまう，実行履歴の量が膨大であり，情報量が多くなる，実行できる単位でしか解析できない，という欠点もある．

本研究では，動的解析からのシーケンス図の作成を目指しており，上記の幾つかの問題点のうち，実行履歴の量に関する問題を緩和しようと試みている．これより，特定のテストケースによるプログラムの実行から，理解しやすいシーケンス図を提示することができ，保守作業やデバッグ作業における「プログラム中のオブジェクトがどのように動作するかを理解する」作業を支援することができる．

3　実行履歴圧縮手法

本章では，実行履歴の取得方法やデータ構造，および実行履歴の圧縮手法の詳細と，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．

3.1　動的解析による実行履歴の取得

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

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

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

実際に取得した実行履歴を図 2 に示す．図中の<init>はコンストラクタを表している．この実行履歴では，プログラムが amida パッケージのクラス Main の static メソッド main から実行され，main の中でクラス MainFrame のオブジェクト 1 に対してメソッド<init>，つまりコンストラクタの呼び出しが行われる．さらにその中で，クラス SearchDialog の 2 番のオブジェクトに対してコンストラクタの呼び出しが行われ，次に，クラス MainFrame の static メソッドであるメソッド getInstance が呼ばれている．

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

図 2: 実行履歴の例

表 1: 記法の定義

表記	意味
x	任意の頂点
x.object	頂点 x のオブジェクトの ID
x.method	頂点 x のメソッドのシグネチャ
x.callNum	頂点 x の子の数 (x.callNum ≥ 0)
x[i]	頂点 x の i 番目の子である頂点 ($1 \leq$ i $\leq$ x.callNum)
S	部分木の列
S.size	部分木の列 S に含まれる部分木の個数
S[i]	部分木の列 S に含まれる i 番目の部分木 ($0 \leq$ i $\leq$ S.size)

3.2 圧縮ルール

3.1 節で取得した実行履歴中には，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．そのため，実行履歴は膨大な量に上ることが多い [15]．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 [9]．そこで，本手法では，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．

繰り返し構造を圧縮する方法として R1 から R3 の 3 つのルールを考案し，また，再帰構造の圧縮方法として R4 を考案した．

それぞれのルールについて説明するために，実行履歴のメソッド呼び出し構造を木構造に置き換えたものを用いる (図 3)．各メソッド呼び出しを頂点とし，それぞれの頂点は，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，その頂点の子として辺を引く．例えば，図 3 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 3 において，実行履歴中の最初のメソッド呼び出しである，クラス A のオブジェクト 1 へのメソッド a() の呼び出しを表す頂点を x と置く．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 となり，それぞれのメソッド呼び出しを表す頂点は x[1], x[2] で表される．同様に，x[1].callNum = 0, x[2].callNum = 1, であり，x[2] の根である頂点から呼ばれるメソッド d の呼び出しを表す頂点は x[2][1] となる．

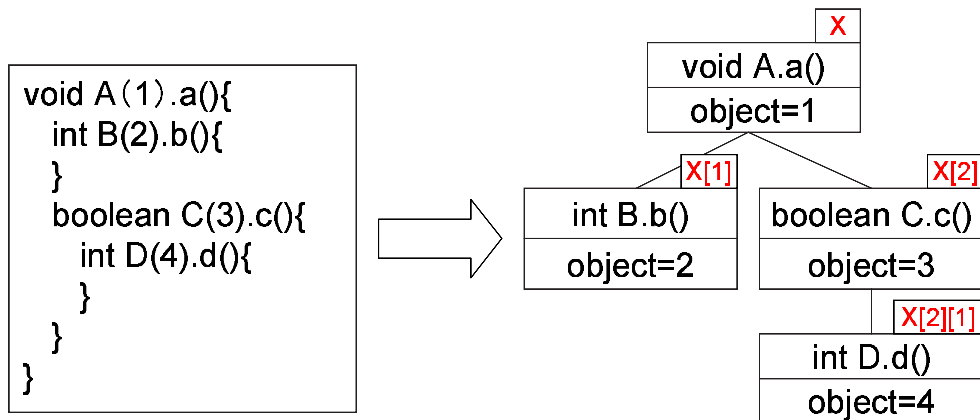

図 3: 木構造と記法の例

3.2.1 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．

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は，まず，ある頂点 x について，その子である頂点を根とする部分木の列から,「同一」な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 回辺りの部分木の個数が x.callNum/2 以上に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は，繰り返しの回数も記憶しておく．どのようなものが「同一」であると判定されるかについては，各ルールによって異なる．

以上の流れを図で表したものが図 4 である．図中の 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」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 回辺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 同士は同一であると判定され，その次の t1 は同一ではないから (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

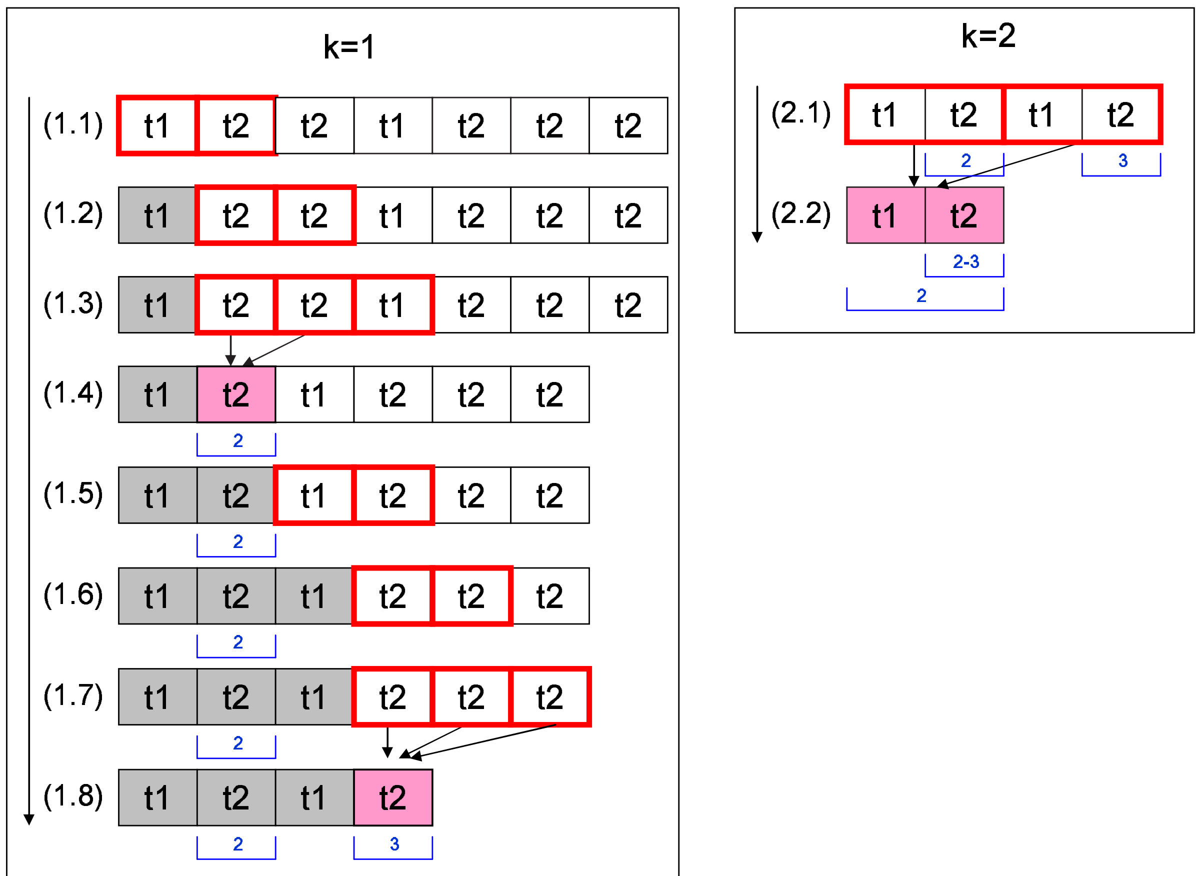

図 4: 繰り返し構造の圧縮処理の流れ

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，繰り返しを終える．このような流れで処理が進められていく．

各繰り返し圧縮ルールは部分木の同一性の判定において，子以下も含めた比較を行っていくが，ここであらかじめ定数 D が定められていて，部分木の根から D 階層下にある頂点までしか比較しないこととし，D 階層以内の木構造を比較範囲と呼ぶ．つまり図 5 に示す二つの部分木を比較する場合，D=2 のときは比較対象の部分木の根とその子までが比較範囲になる．

以下では，各圧縮ルールの処理について説明する．

R1:完全な繰り返し

圧縮ルール R1 では実行履歴中から，完全に同一な呼び出し構造が繰り返されている

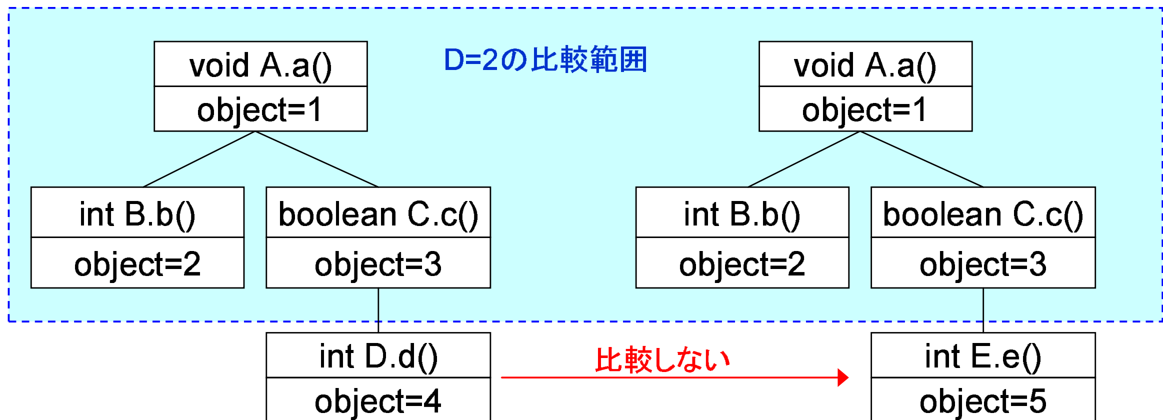

図 5: 定数 D における比較範囲

箇所を検出し，圧縮する．つまり，図 4 を用いて説明した処理の中で，同一であるかどうかの判定において，比較範囲内の木構造が同様の構造であり，かつ，それぞれの頂点のメソッドとオブジェクトが等しいものを同一と判定する．また，置き換えの対象となる，繰り返し全体を現す部分木は，繰り返し 1 回目の部分木をそのまま用いることとする．

R1 において同一性を判定する処理を記述したものを図 6,7 に示す．図 6 中の treesEquals 関数は 2 つの頂点の列を引数に取り，各頂点を根とする部分木について同一性の判定を行う．つまり，treesEquals 関数は図 4 の太枠で囲まれた部分 2 つの比較を行うということになる．そして，treesEquals 内で呼ばれる図 7 の treeEquals 関数が部分木 1 つ分の比較を行う．図 4 の (1.3) や (1.7) のように繰り返しを検出した場合，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)．

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．また，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，同一であるかどうかの判定において，比較範囲内の木構造が同様の構造であり，かつ，それぞれの頂点のメソッドが等しいものを同一と判定する．処理の内容自体は R1 とほぼ同様であるが，部分木を比較する際に，objectID の比較を行わない部分が異なる．そして，繰り返しと置き換えられる，繰り返し全体

```
boolean treesEquals(S1,S2){
  if (S1.size != S2.size)
    return false;

  for (i = 1; i <= S1.size; i++){
    if (! treeEequals(S1[i],S2[i],D))
      return false;
  }
  return true;
}
```

図 6: R1,R2 における部分木列の比較処理

```
boolean treeEquals(x,y,depth){
  if (x.method != y.method || x.object != y.object){
    return false;
  if (depth == 1)
    return true;
  if (x.callNum != y.callNum){
    return false;

  for(i = 1; i <= x.callNum; i++){
    if (! treeEquals(x[i],y[i],depth-1))
      return false;
  }
  return true;
}
```

図 7: R1 における部分木の比較処理

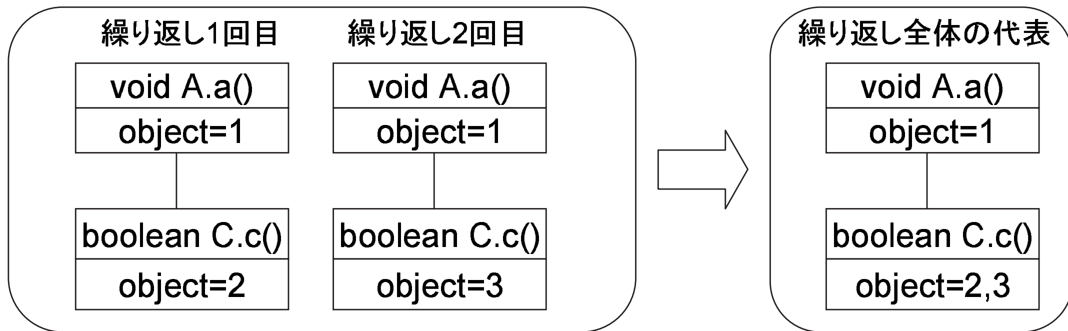

図 8: R2 における繰り返しの置き換え

```
boolean treeEquals(x,y,depth){
  if (x.method != y.method){
    return false;
  if (depth == 1)
    return true;
  if (x.callNum != y.callNum){
    return false;

  for(i = 1; i <= x.callNum; i++){
    if (! treeEquals(x[i],y[i],depth-1))
      return false;
  }
  return true;
}
```

図 9: R2 における部分木の比較処理

を現す部分木列は，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，置き換えを行う (図 8)．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．

処理の詳細としては，図 7 中の treeEquals の代わりに，図 9 の treeEquals を用いる．

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欠損，及び包含関係について定義する．呼び出し構造の欠損とは図10のように，ある木構造 t_1,t_2 があり，その t_1 からいくつかの部分木を取り除いたものが t_2 と同一である時，t_2 には t_1 から取り除かれた部分木が欠損しているとみなし，t_1 が t_2 を包含しているといい，$t_1 > t_2$ と表す．また，ここで同一であるとは，定数Dによって指定される比較範囲において同様の木構造を持ち，その範囲内の各頂点のメソッドがそれぞれ等しいことであるとする．オブジェクトIDについてはここでは考慮しない．すなわち，部分木 t_1,t_2 が同一であるとは，それぞれの部分木の根である頂点を x_1,x_2 とすると，図9におけるR2用の treeEquals(x_1,x_2,D) が true であることとし，その関係を $t_1 = t_2$ で表すとする．さらに包含関係を拡張して，任意の部分木列 S_1,S_2 が $S_1 > S_2$ であるとは，S_1.size $=$ S_2.size and S_1[i] $>$ S_2[i](∀i, 1 ≤ i ≤ S_1.size) であり，$S_1 \geq S_2$ とは，S_1.size $=$ S_2.size and (S_1[i] $>$ S_2[i] or S_1[i] $=$ S_2[i])(∀i, 1 ≤ i ≤ S_1.size) であるとする．

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4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2つの部分木列を S_1,S_2 とおいた時に，$S_1 \geq S_2$ または $S_1 < S_2$ 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t，繰り返しになっている部分木列を S_1, ... ,S_t とし，次の比較対象の部分木列を S_{t+1} とすると，$S_i \geq S_j(\forall j, 1 \leq j \leq t)$ を満たす S_i について，$S_{t+1} \geq S_i$ または $S_{t+1} < S_i$ である時同一であるとする．

この繰り返しと置き換えられる部分木列は，繰り返し中の最大の要素，つまり，繰り返しの回数をtとすると，$S_i \geq S_j(\forall j, 1 \leq j \leq t)$ を満たす S_i を基にして，R2の時と同様のオブジェクトの統合を行い，また，S_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．

部分木，及び部分木列の大小関係を判定する処理の詳細を図11,12に示す．図中の2つの関数は，それぞれ引数として受け取った部分木列や部分木について，第1引数=第2引数なら0を，第1引数 > 第2引数なら1を，第1引数 < 第2引数なら2を，それ以外なら-1を返す関数である．

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，元の実行履歴の構造を正確には表していないことになる．

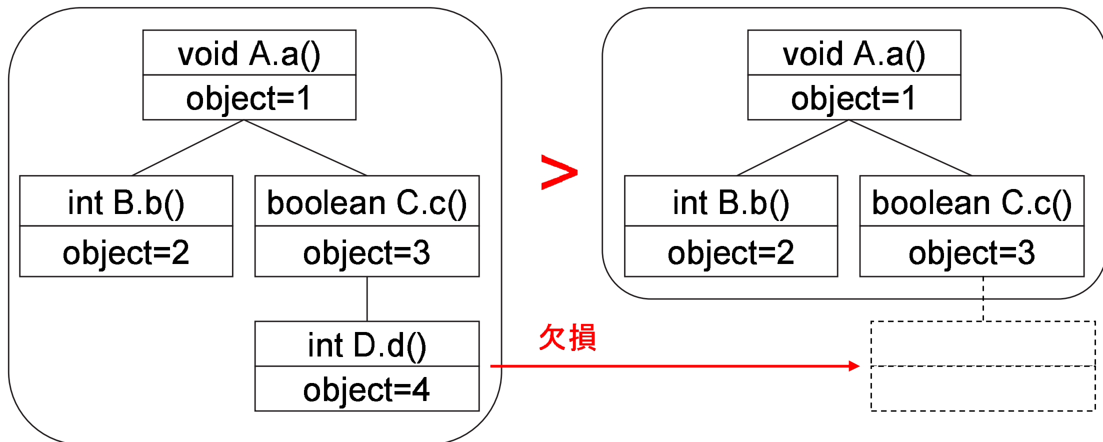

図 10: 欠損構造の例

```
/*
返り値      : S1=S2なら0, S1>S2なら1, S1<S2なら2,
              それ以外なら-1
*/
int treesCompare(S1,S2){
  if (S1.size != S2.size)
    return -1;

  int result = 0; //その段階までの比較結果(0,1,2のいずれか)

  for (i = 1; i <= S1.size; i++){
    compare = treeCompare(S1[i],S2[i],D)
    if (compare == -1)
      return -1;
    else if (compare != 0){
      if (result == 0)
        result = compare;
      else if (compare != result)
        return -1;
    }
  }
  return result;
}
```

図 11: R3 おける部分木列の比較処理

```
/*
返り値:x=yなら0, x>yなら1, x<yなら2, それ以外なら-1
*/
int treeCompare(x,y,depth){
  if (x.method != y.method){
    return -1;
  if (depth == 1)
    return 0;

  int result = 0; //その段階までの比較結果(0,1,2のいずれか)
  if (x.callNum > y.callNum)
    result = 1;
  else if (x.callNum < y.callNum)
    result = 2;

  int i = 1;
  int j = 1;
  while(i <= x.callNum && j <= y.callNum){
    compare = treeCompare(x[i],y[j],depth-1))
    if (compare == 0 || compare == result)
        i++,j++;
    else if (compare == -1){
      if (result == 0)
        return -1;
      else if (result == 1)
        i++;
      else
        j++;
    }
    else if (result == 0){
      result = compare;
      i++; j++;
    }
    else
      return -1
  }

  if (i == x.callNum && j == y.callNum)
    return result
  else if (i != x.callNum && result != 2)
    return 1;
  else if (j != y.callNum && result != 1)
    return 2;
  else
    return -1;
}
```

図 12: R3 における部分木の比較処理

表 2: R4 で用いる記法の定義

表記	意味
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頂点
replace(x,y)	頂点 x を親から切り離し，その場所に頂点 y を繋げる
x.tree	x を根とする部分木
T	頂点の列
T.size	頂点の列 T に含まれる頂点の個数
T[i]	頂点の列 T に含まれる i 番目の頂点 ($1 \leq i \leq$ T.size)

3.2.2 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頂点を x とし，圧縮処理の詳細を 4 段階に分けて説明する．

1. 再帰的に呼ばれているメソッドを表す頂点の特定
   メソッド呼び出しの順序に従って木構造を探索し，葉である頂点 y を見つける．もし，それ以前にすでに，y を葉として以降の処理を行っていた場合は，次の葉へ進む．そうでなければ，x から y へ辿る道上に存在する各頂点の列 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) を満たす最小の 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] の個数を k 個 (k$\geq$2) とし，それらの

列を $\acute{T}$ とする．つまり，$\acute{T}$ 中の各頂点は，T 中に含まれる頂点のうち，最初に現れる，再帰的に呼ばれるメソッド呼び出しを表す頂点である．$\acute{T}$ 中の全ての頂点が，すでに再帰呼び出し構造として圧縮処理を受けている場合は次の葉へ進む．

2. 再帰呼び出しの頂点の切り離し (copy への置換)
   次に，$1 \leq$ i $\leq$ k を満たす全ての i について，$\acute{T}$[i].parent が存在するなら，replace($\acute{T}$[i],$\acute{T}$[i].copy) を行う．つまり，頂点 $\acute{T}$[i] をその親から切り離して，代わりにその位置に頂点 $\acute{T}$[i] と同じ ID と同じメソッドの頂点をおく．ここで，$\acute{T}$[1] が x であった場合は parent が存在しないことがある．その場合は，$\acute{T}$[1].copy には置換しない．また，各 $\acute{T}$[i].copy の列を C とする ($\acute{T}$[1].copy = C[1])．

3. 代表元となり得る頂点の選別
   次に，R3 の説明で定義した包含関係を用いて，$\acute{T}$ 中から代表になる頂点を選ぶ．ここで，x は y に包含されないことを，x ! < y，x=y でないことを，x ! = y と記述することにする．$\acute{T}$[i].tree ! < $\acute{T}$[j].tree ($\forall$j,1 $\leq$ j $\leq$ k) 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．このとき，$\acute{T}$ 中の頂点のうち，これら以外のものは，これらの m 個の頂点のいずれかに包含されるか等しいものになる．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，$\acute{T}[i_1]$, $\acute{T}[i_2]$, ... , $\acute{T}[i_m]$ の各頂点と，C 中の各頂点 C[i_1+1],C[i_2+1], ... ,C[$i_{m-1}+1$] について，今回，再帰呼び出しとして圧縮処理を受けたことを記録し，それら全てのオブジェクト ID を統合する．

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．C[1].parent がない場合は，x が $\acute{T}[i_1]$ でないなら，圧縮対象の部分木全体を $\acute{T}[i_1]$.tree と置き換える．次に，j $\geq$ 2 ならば，全ての 2 $\leq$ j $\leq$ 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．もし，y が $\acute{T}[i_m]$.tree に含まれていれば，葉 y についてもう一度同じ処理を行う．含まれていなければ $\acute{T}[i_m]$.tree 中の最初に到達する葉に進む．

以上のような再帰構造の圧縮処理の例を図 13 に示す．

この例では，図の上段の木構造の，クラス A のオブジェクト 1 に対するメソッド a の呼び出しを表す頂点を x とし，x 以下の部分木が圧縮処理を受ける対象の部分木であるとする．

まず初めに検出される葉 y は x[1][1][1] のオブジェクト 4 へのメソッド b の呼び出しである．この時，T={x,x[1],x[1][1],X[1][1][1]} には，x.method = x[1].method が成り立

つので，t = 1 であり，x.method = T[r].method($1 \leq$ r ≤ 4) を満たす r は {1,2,3} であるから，k = 3 で，$\acute{T}$ = {x,x[1],x[1][1]} である．x,x[1],x[1][1] はいずれも，再帰呼び出しとして処理済みではないから，処理を進めていく．

まず，$\acute{T}$[1].copy すなわち，x.copy を作るが，x.parent は存在しないので，x.copy を C に入れる．次に，$\acute{T}$[2].copy すなわち，x[1].copy を作り，replace($\acute{T}$[2],$\acute{T}$[2].copy)，つまり，replace(x[1],x[1].copy) を行い，x[1] 以下の部分木を x から切り離し，x[1].copy が新しい x[1] になる．ここで，切り離された方を以降 ($\acute{x[1]}$) と呼ぶこととする．そして，C にこの x[1].copy を入れておく．同様に，replace(x[1][1],x[1][1].copy) を行い，切り離されたものを ($\acute{x[1][1]}$) とする．

ここまでの処理を行ったのが，図 13 の中段の図である．

次に，R3 で定義した包含関係を用いて，再帰構造を代表する頂点の候補を選ぶ．まず，$\acute{T}$[1].tree ! < $\acute{T}$[2].tree かつ $\acute{T}$[1].tree ! < $\acute{T}$[3] であり，$1 \leq$ j < 1 なる j は存在しないので $\acute{T}$[1] は条件を満たす．$\acute{T}$[2] については，$\acute{T}$[2].tree = $\acute{T}$[1].tree であるので，条件を満たさない．$\acute{T}$[3] も，$\acute{T}$[2].tree > $\acute{T}$[3].tree であるので，条件を満たさない．よって，m=1 であり，{$\acute{T}[i_1]$, $\acute{T}[i_2]$, ... , $\acute{T}[i_m]$} = {x} である．ここで，条件を満たせなかった $\acute{T}$[2].tree と $\acute{T}$[3].tree から，オブジェクトや欠損構造の情報が付加される．つまり，($\acute{x[1]}$).object の 2 と，($\acute{x[1][1]}$).object の 3 が x.object に追加される．同様に，($\acute{x[1]}$)[1].object の 3 が x[1].object に，($\acute{x[1]}$)[2].object の 5 と，($\acute{x[1][1]}$)[1].object の 4 が x[2].object にそれぞれ反映され，x[1] は欠損する可能性を付加される．その後，$\acute{T}$[1] である x と C[2] である x[1] に再帰呼び出しとして処理済であることを記録し，それぞれの object を統合する．つまり，x.object={1,2,3} であり，x[1].object={2,3} であるから，x[1].object に 1 が追加される．図中では欠損の可能性のある部分木を辺に “?” を，再帰処理済の頂点を “R” を付けて表している．

そして最後に，再構成を行うが，C[1] は parent を持たず，$\acute{T}[i_1]$ は x なので，そのまま終了となる．結果は図中の下段の木になる．

$\acute{T}[i_m]$.tree である x.tree 中に葉 y は無いので，$\acute{T}[i_m]$.tree 中の末端を順に探索する．まず，x[1] に到達する．T={x,x[1]} であり，$\acute{T}$ = {x,x[1]} となるが，x と x[1] はどちらも再帰処理済みであるから次の葉へ進む．次は，x[2] に到達する．T = {x,x[2]} であるが，T[t].method = T[r].method となる t,r が存在しないので，次の葉へ進む．葉がなくなったので，圧縮処理を終了する．

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．

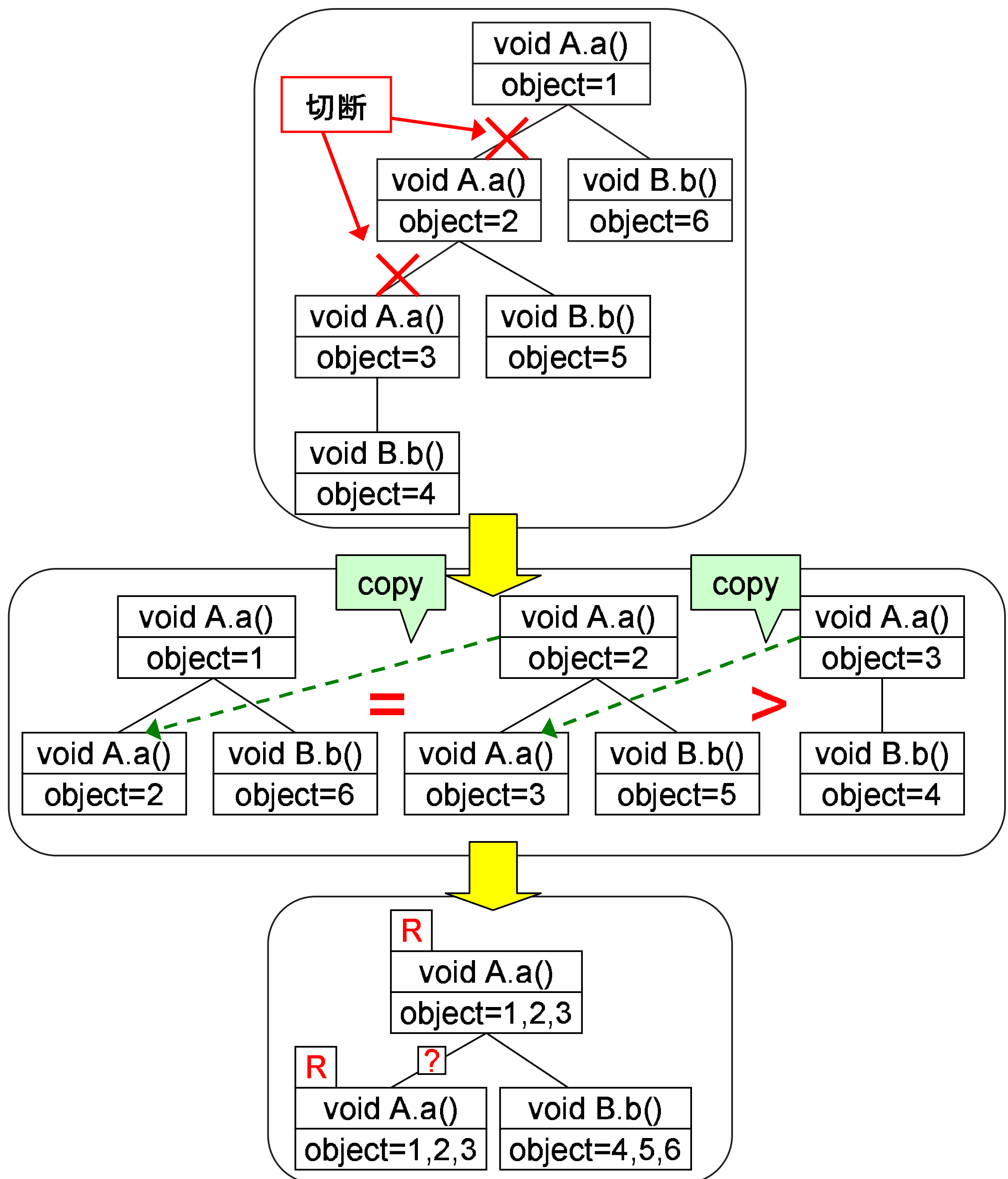

図 13: 再帰構造の圧縮例

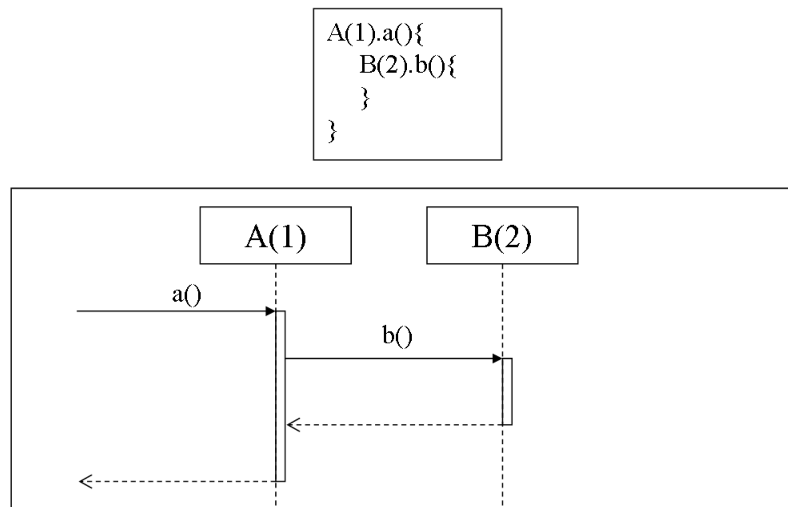

図 14: 未圧縮部から作成される図

3.3 シーケンス図生成への適用

3.2 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．

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 14).

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

R2 被圧縮部

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

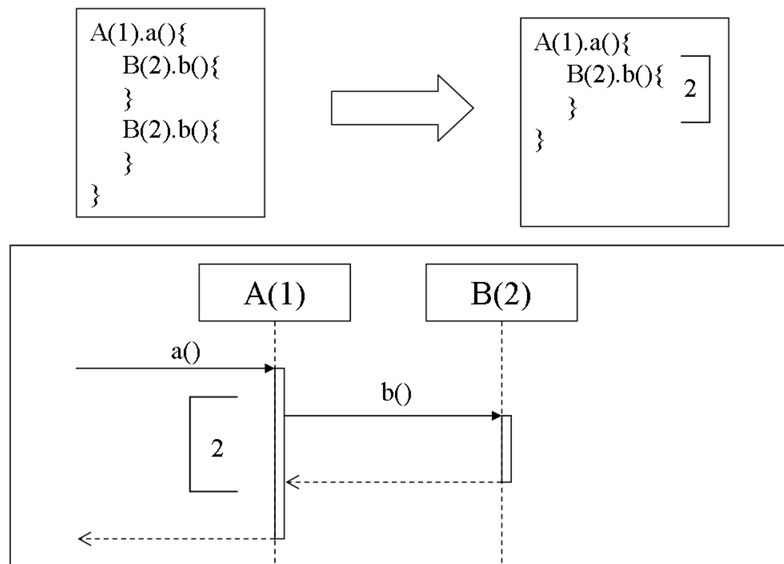

図 15: R1 適用部から作成される図

ジェクトを追加し，それに対するシーケンスを引く (図 16)．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 1 と 2 が統合され，別の繰り返しの圧縮により，オブジェクト 2 と 3 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 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 17)．

R4 被圧縮部

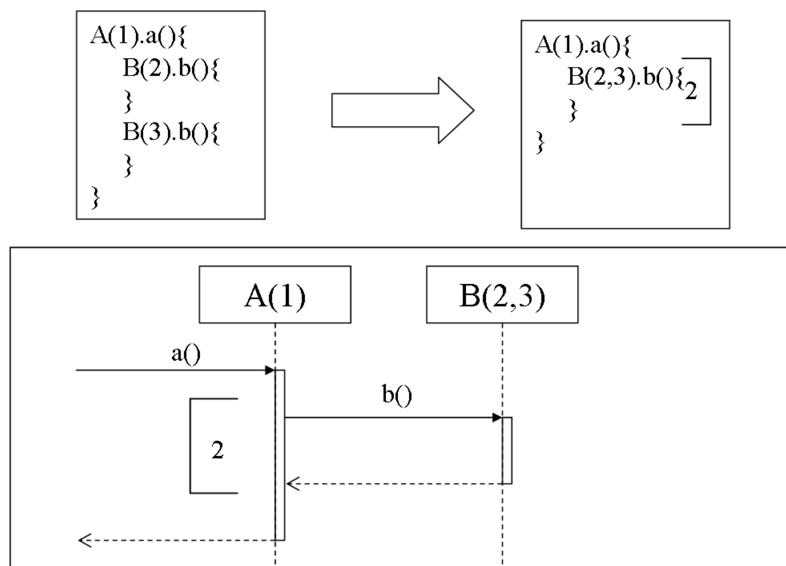

図 16: R2 適用部から作成される図

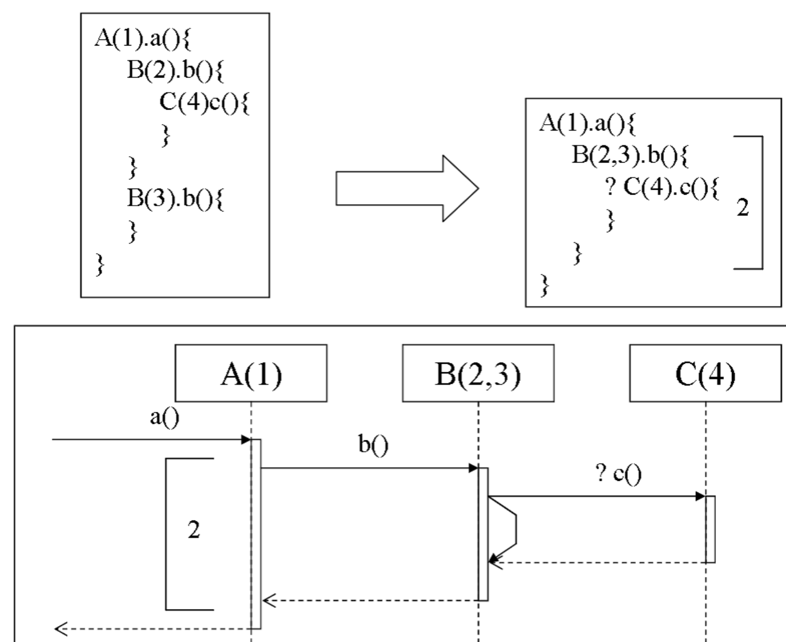

図 17: R3 適用部から作成される図

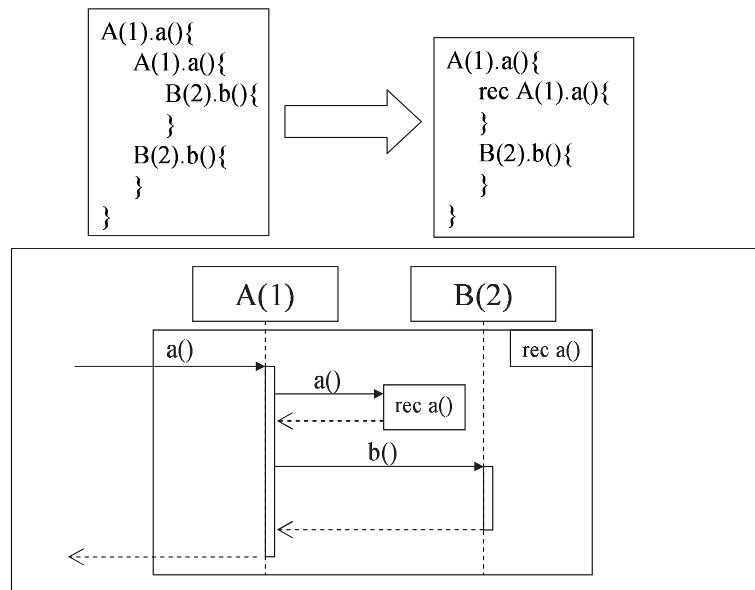

図 18: R4 適用部から作成される図

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 18).

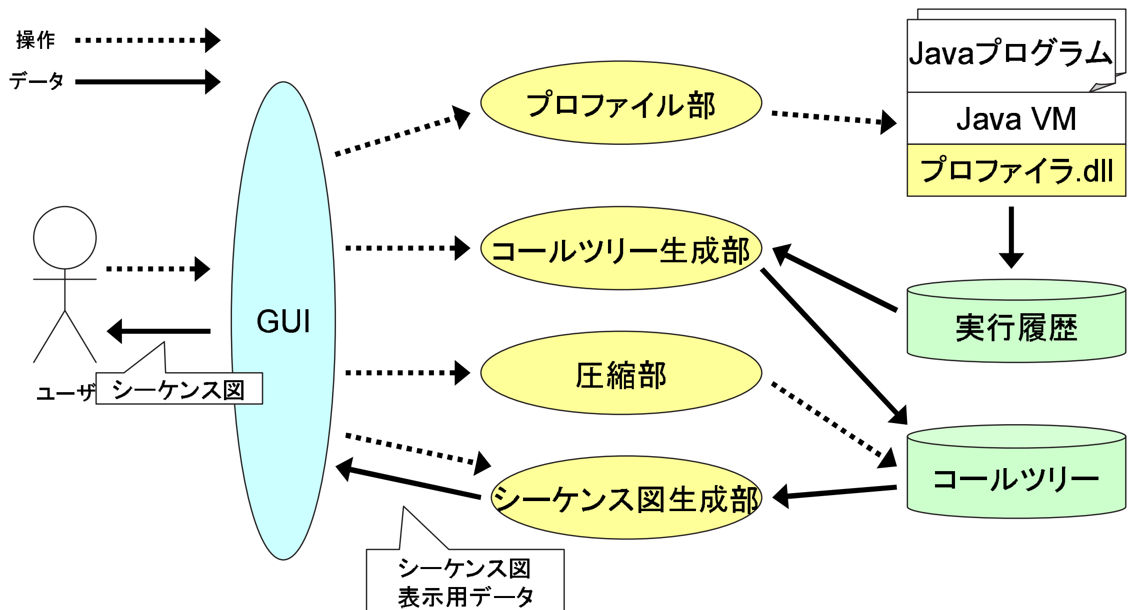

図 19: ツールの構成図

4　ツールの実装

4.1　ツールの実装

3 章において提案した手法をツールとして実装を行った．実装には概ね Java 言語を用いた．ただし，後述するプロファイラ DLL については C++言語で実装を行った．ソースコードの量は Java 言語の部分が約 15,000 行，C++言語の部分が約 1,000 行であった．

4.2　ツールの構成

本ツールは，GUI，プロファイル部，プロファイラ DLL, コールツリー生成部，シーケンス図生成部，圧縮部の 6 つのコンポーネントから構成されている．ツールの構成図を図 19 に示す．

図 19 上の点線の矢印は操作の流れを表し，実践の矢印はデータの流れを表している．以下，各コンポーネントについて詳細を述べる．

4.2.1　GUI

GUI は大きく 2 つに分けられる．他コンポーネントを操作する部分とシーケンス図を表示する部分である．他コンポーネントを操作する部分については各コンポーネントについての節で述べる．シーケンス図表示部は図 20 のオブジェクト表示部，シーケンス表示部，縮

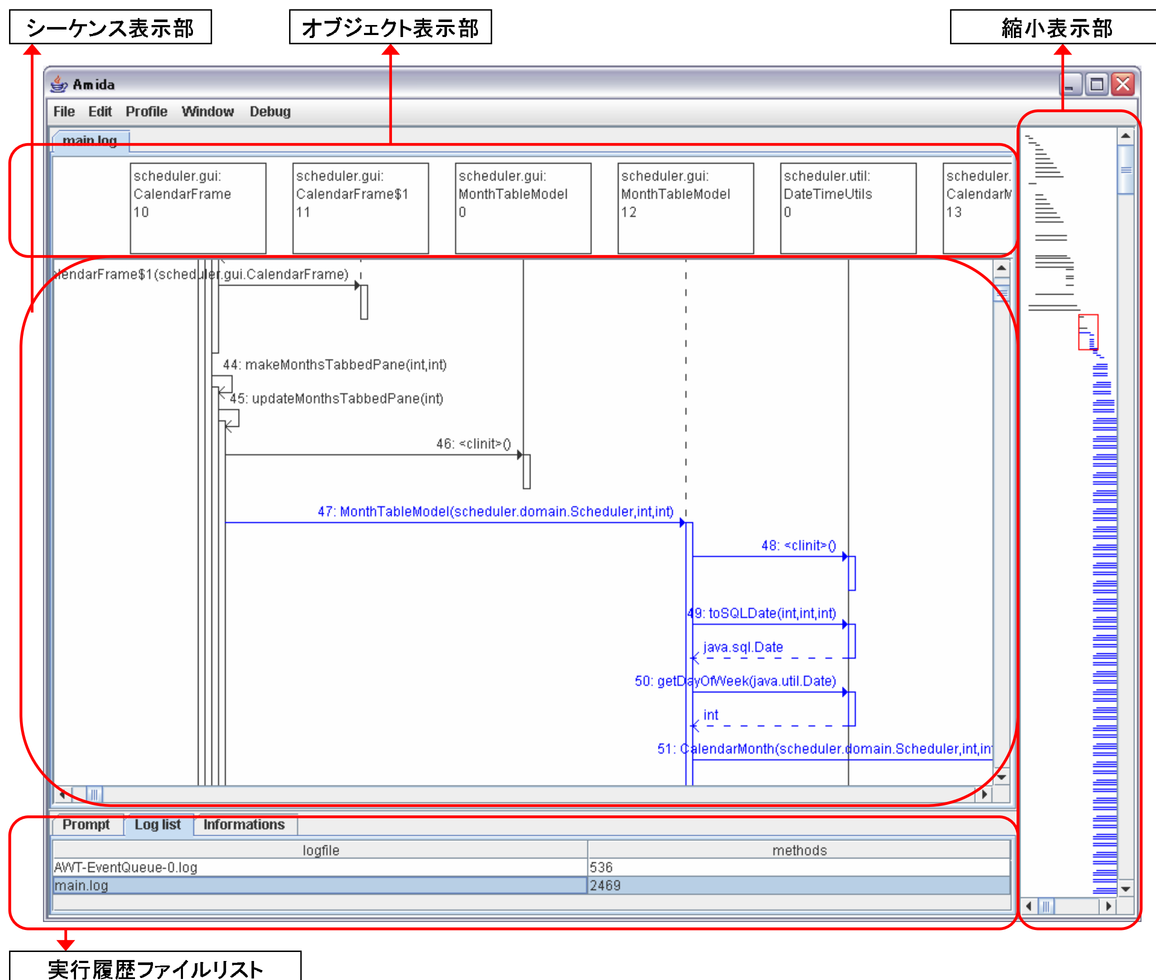

図 20: メイン画面

小表示部の 3 つから構成されている．縮小表示部とはシーケンス図全体を縮小表示する部分で，他の 2 つの部分が表示している範囲を赤枠で示すことで，シーケンス図全体の中でどの辺りを表示しているのかが分かるようにしている．シーケンス図表示部はタブ形式になっており，シーケンス図生成部から受け取った 1 つのシーケンス図表示用データに対して，1 つのタブでシーケンス図を表示する．

ユーザは表示されたシーケンス図上で，圧縮された部分を選択して展開したり，選択したメソッド呼び出し以降を圧縮処理に掛けたり，一部を切り離して，別のシーケンス図として表示させることができる．また，Edit メニューの Search からメソッドやオブジェクトを検索することもできる．

4.2.2 プロファイラ DLL

プロファイラ DLL は VM と一緒に動作し，実行履歴を取得する部分である．Sun の Java Virtual Machine(VM) に用意されている Java Virtual Machine Profiler Interface (JVMPI)[8] を利用し，C++言語で記述されたダイナミックリンクライブラリとして実装を行った．このプロファイラ DLL を VM にコマンドラインオプションとして渡して解析対象となる実行プログラムを実行することで，実行プログラムで発生する各メソッド呼び出しについて，3.1 節で述べた情報をスレッドごとに実行履歴ファイルに記録できる．

また，プロファイラ DLL には実行履歴を取得する際に設定ファイルに記述することで，以下のような機能を利用して，どのように実行履歴を取得するかを指定することが出来る．

- 実行履歴ファイルを保存するディレクトリの指定．

- 実行履歴の取得を開始するメソッドの指定．
  プロファイラ DLL は指定したメソッドの呼び出しとその内部で発生したメソッド呼び出しのみを実行履歴として保存する．

- 実行履歴を取得対象とならない，パッケージ，クラス，メソッドの指定．
  プロファイラ DLL はここで指定されたパッケージ，クラスのオブジェクトに対するメソッド呼び出しや，指定したメソッドの呼び出しを記録しなくなる．ただし，次の機能を利用し，その条件に該当する場合はその限りではない．

- 取得対象のメソッド内で発生した取得対象とならないメソッド呼び出しを記録するかどうかの指定．
  取得対象であり注目したいメソッドのソースコード内において取得対象とならないメソッド呼び出しが記述されている場合，ソースコード内に記述されているメソッド呼び出しが実行されても，実行履歴には記録されないという事態が発生する．そのような場合に，この機能を使うことでソースコードに記述されているメソッド呼び出しを全て記録することができる．具体的には，ライブラリパッケージの実行履歴は無視したいが，ユーザプログラム内に記述されたライブラリパッケージへのメソッド呼び出しは記録したい場合などに使用する．

- ログファイルの記録形式の指定．
  通常は図 2 のようなテキスト形式で記録するが，サイズを大きくしたくない場合には，各メソッドに ID を付け，各メソッド呼び出し毎に，呼び出しを受けたオブジェクトの ID とメソッドの ID の対をバイナリ形式で記録する．

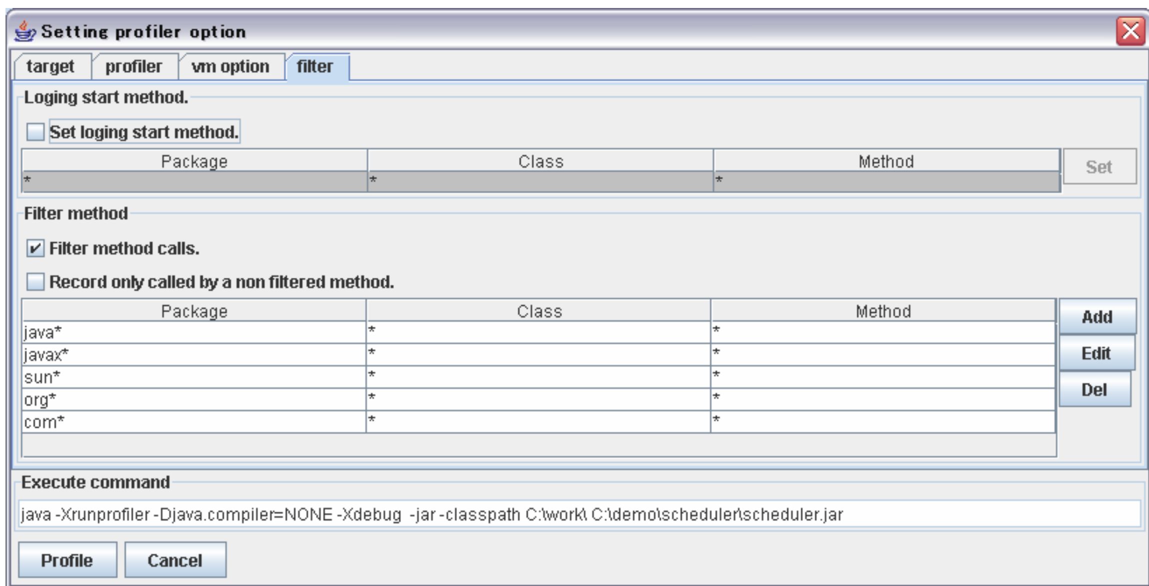

図 21: プロファイラ設定画面

本ツールは，現在のところ，Java 言語で書かれたプログラムのみを対象としているが，言語および実行系に依存しているのはこのプロファイラ DLL のみであり，他のオブジェクト指向言語と実行系を対象とした実行履歴取得システムを作成することにより，他言語への拡張も可能である．

4.2.3　プロファイル部

プロファイル部は対象プログラムを VM 上で実行する操作を行う．具体的には，GUI のプロファイル設定画面 (図 21) で指定された設定に従って，実行履歴を取得する対象の Java プログラムとプロファイラ DLL 決定し，プロファイラ DLL の設定ファイルを作成後，指定されたオプションを用いて VM 上でプログラムを実行する．

4.2.4　コールツリー生成部

コールツリー生成部はプロファイラ DLL が取得した実行履歴ファイルを読み込み，メソッド呼び出し構造をツリー形式のデータ構造に変換する．GUI からコールツリー生成部を操作するには 2 通りの操作方法がある．1 つは File メニューの Load から実行履歴ファイルを指定し，コールツリー生成部にファイルを読み込ませる方法である．対象プログラムの直前の実行で記録された実行履歴ファイルを読み込ませる場合は，2 つめの方法である実行履歴ファイルリスト (図 20) から選択して読み込ませることもできる．

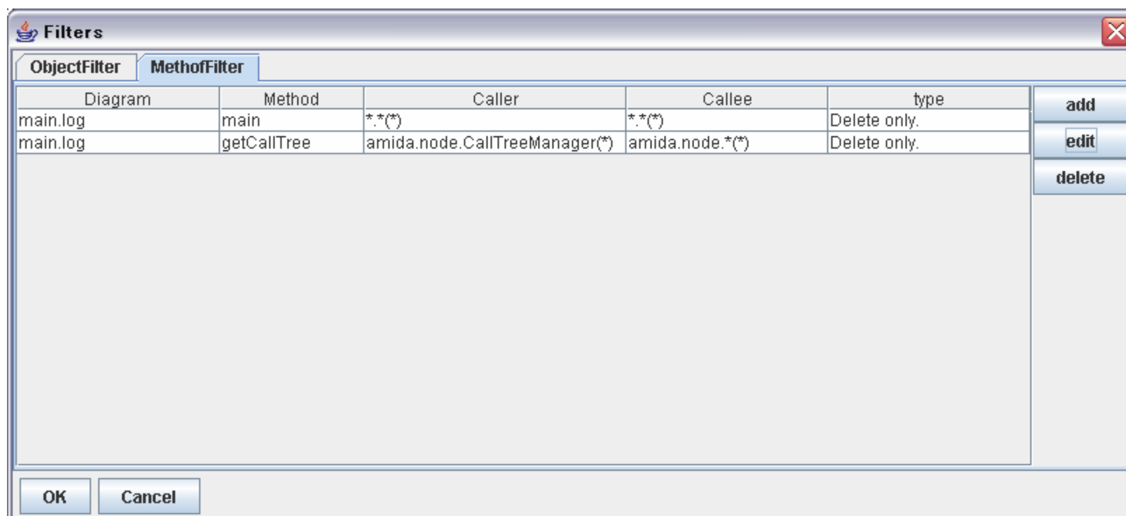

図 22: フィルタリング設定画面

4.2.5　シーケンス図生成部

シーケンス図生成部は，コールツリーの任意の部分木を基に 3.3 節の方針にしたがってシーケンス図表示用のデータを生成する．基本的には，コールツリー生成部がコールツリーを生成するたびに，自動的にそのコールツリー全体を表すシーケンス図表示用のデータを生成し，GUI のシーケンス図表示部にデータを送る．ユーザは，GUI のシーケンス図表示部に表示されたシーケンス図中から，任意の範囲を別の図として切り離すことが出来る．この操作が行われた場合，シーケンス図生成部は切り離された部分に対応するコールツリーの部分木から，新たなシーケンス図表示用データの作成を行う．

また，シーケンス図生成部はフィルタリング機能を持っており，パッケージ名やクラス名などの特定の条件を満たすオブジェクトを 1 つのオブジェクトとして統合する，図上には表示しなくする，などが可能である．メソッドに対しても，メソッド名，呼び出し先のオブジェクト，呼び出し元のオブジェクトなどの特定の条件を満たすメソッドやそのメソッド呼び出し内で発生する全てのメソッドを非表示にすることができる．これらの機能を利用することで，利用しているライブラリのオブジェクトを 1 つに統合したり，特定のクラス内の内部メソッド呼び出しを非表示にすることができ，利用者の目的に応じて簡潔な図に加工していくことが可能である．ユーザは GUI のフィルタリング設定画面を用いることでこの機能を利用することができる．

4.2.6 圧縮部

圧縮部では 3.2 節で述べた圧縮ルールに従って，任意のコールツリーの任意の部分木に対して圧縮処理を行う．ユーザは Edit メニューの Compression から任意の圧縮ルールを選択することで，現在表示中のシーケンス図に対応するコールツリーの部分木に圧縮処理を掛けることができる．また，表示されたシーケンス図から特定のメソッド呼び出しを選択し，そのメソッド呼び出し以下のシーケンスに対応するコールツリーの部分木に対して，圧縮処理を掛けることも可能である．

4.3 ツールの利用手順

ユーザは，本ツールの GUI を用いて対話的に分析を行う．まず，ユーザは解析対象とする実行プログラムと，そのプログラムを動作させる入力データを用意する．次に，本ツールを起動し，Profile メニュープロファイラ設定画面 (図 21) を呼び出す．次に，プロファイラ設定画面から詳細な設定を指定し，対象プログラムを実行する．実行が開始されると VM にオプションとして渡されたプロファイラ DLL が，メソッド呼び出しの実行履歴をファイルに保存していく．

対象プログラムの実行が終了すると，スレッド毎に記録された実行履歴ファイルの一覧が実行履歴ファイルリスト (図 20) に表示され，その中から任意のスレッドを記録した実行履歴ファイルを指定して，コールツリー生成部に読み込ませる．コールツリー生成部は読み込んだ実行履歴を基に，メソッド呼び出し構造を表すツリー構造を構築する．対象プログラムを実行せずに，既存の実行履歴ファイルを読み込む場合は，File メニューの Load から実行履歴ファイルを指定して読み込む．

コールツリーが生成されると，自動的にシーケンス図生成部が動作し，生成されたコールツリー全体からシーケンス図表示用のデータが生成され，GUI のシーケンス図表示部に送られ，それを基にシーケンス図が表示される．

通常，実行履歴全体から生成されたシーケンス図は非常に大きなものになる (図 23) ので，ユーザは次に圧縮操作を行う．Edit メニューの Compression から，表示されているシーケンス図に対応するコールツリーに圧縮操作を行うと，図にその結果が自動的に反映される (図 24)．

また，ユーザは表示されたシーケンス図に対して，様々な操作を行うことが出来る．圧縮された部分を選択して，展開操作を行うことや，オブジェクトを選択して，そのオブジェクトが利用されている箇所を強調して表示することが出来る．また，Edit メニューの Filter からフィルタリング設定画面 (図 22) を呼び出し，フィルタの設定をすることで，シーケンス図生成部にフィルタリング処理を施したデータを新たに生成させることも出来る．その他に

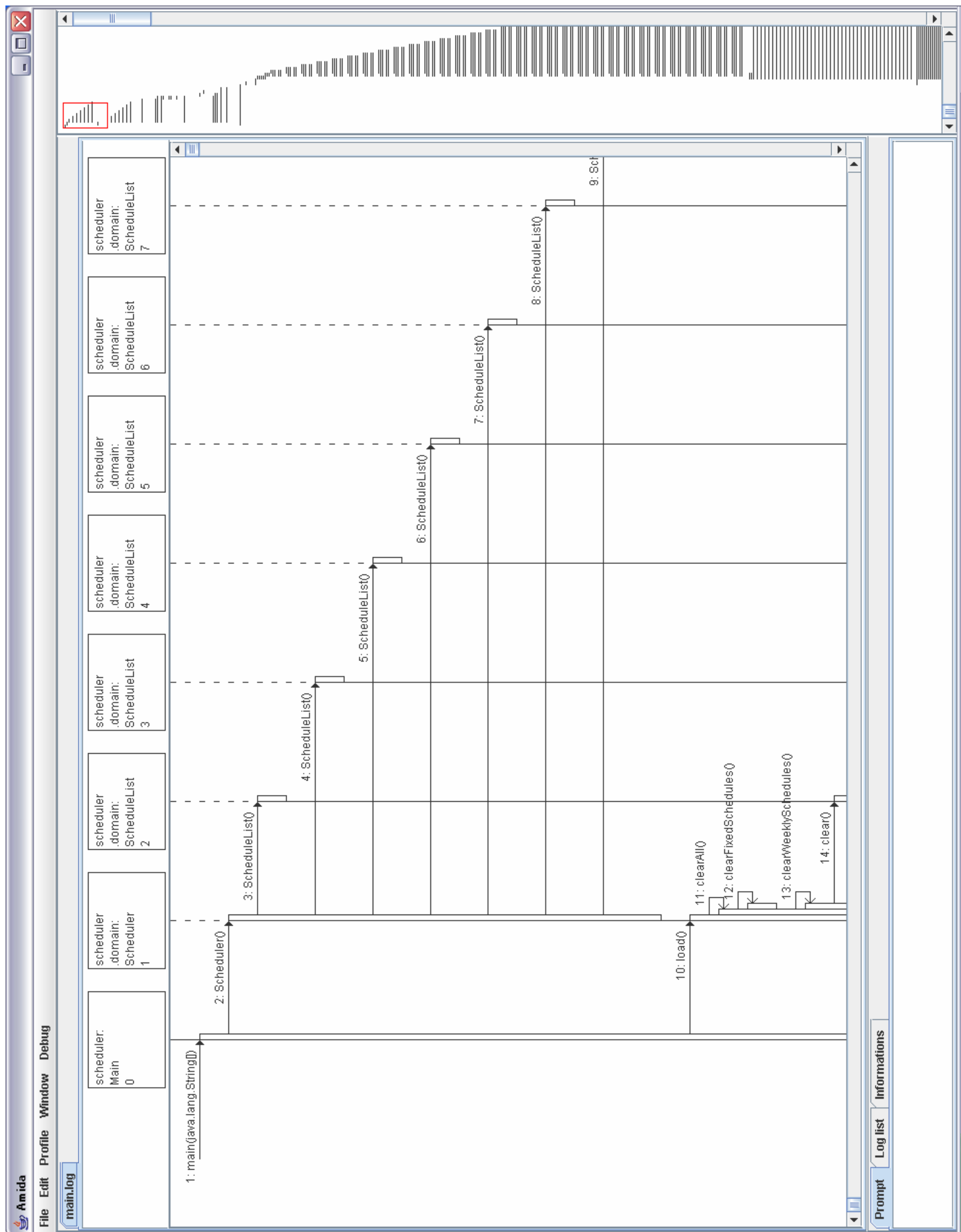

図 23: 圧縮前のシーケンス図

図 24: 圧縮処理後のシーケンス図

も，特定のメソッドやオブジェクトを検索したり，特定のメソッド呼び出し以下のシーケンスを別の図として表示することも可能である．

これらの操作を通じて，ユーザは，圧縮処理により簡潔化されたシーケンス図を用いて，でプログラム実行時の流れを理解することができる．

5　適用実験

3 章で提案した手法について適用実験を行った．本章ではその内容及び結果と，結果に対する考察を述べる．

5.1　実験内容

実験では 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 3 の通りである [23][19][20].

これらのプログラムの起動から各機能に対して，本ツールのプロファイラ DLL を用いて，メソッド呼び出しの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．なお，比較範囲を示す定数 D は 5 とした．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，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，圧縮結果からシーケンス図の生成を行った．特に，scheduler について，設計時に書かれたシーケンス図と生成した図との比較を行った．

5.2　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 および，図 25 から 28 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 の圧縮結果については，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 のように再帰構造が多く現れる実行履歴では，20%程度まで圧縮することができた．

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

表 4: 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

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．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，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 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 ~ 7.22%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%，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，抽象化を行うルールが必要であると思われる．

図 25: jEdit の圧縮結果

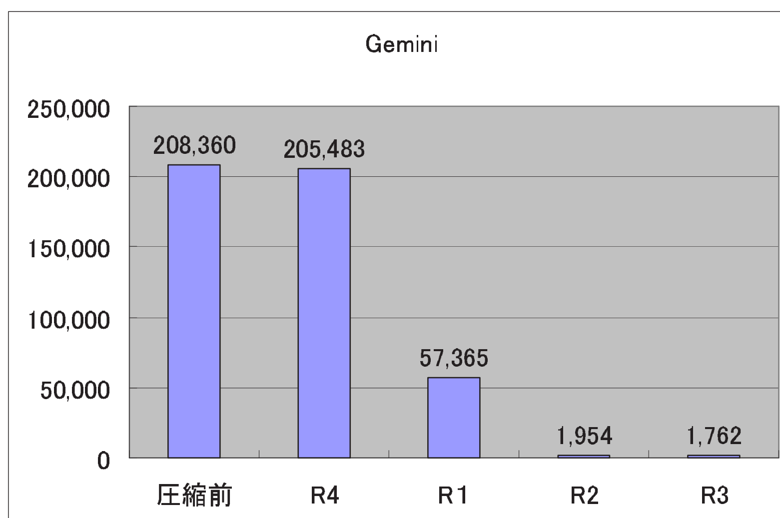

図 26: Gemini の圧縮結果

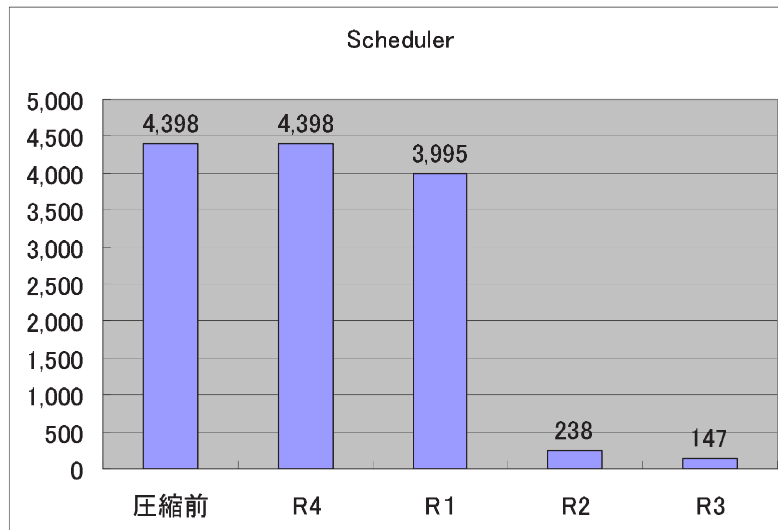

図 27: scheduler の圧縮結果

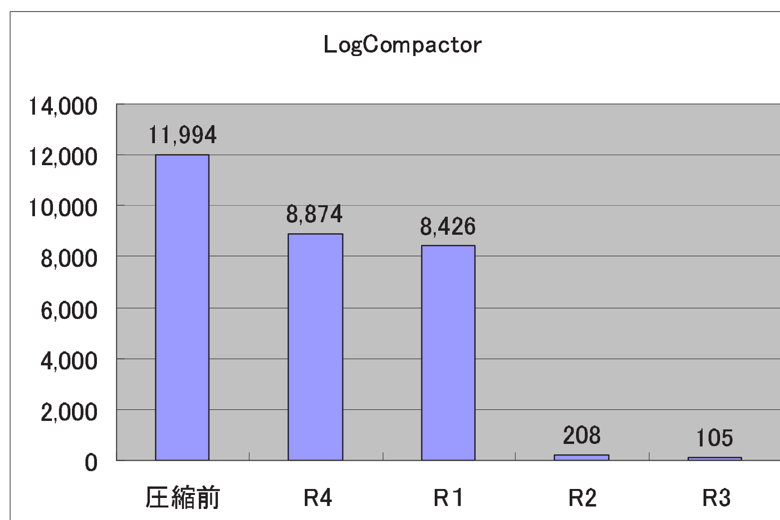

図 28: LogCompactor の圧縮結果

5.3　シーケンス図の生成

次に，圧縮結果からシーケンス図の作成を行った．3.2 節の図 24 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．繰り返しになっている部分やその回数 R2, R3 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図中に表示されている．

最後に，設計時に書かれたシーケンス図と，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った．設計時のシーケンス図を図 29 に，生成したシーケンス図及び比較結果を図 30 に示す．

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 30 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，該当部分のソースコードを確認したところ，設計時のシーケンス図には描かれなかったが，実装の段階で必要になり生成されたオブジェクトや，付け足されたメソッド呼び出しを表していることが分かった．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，これらが出現することは妥当な結果である．

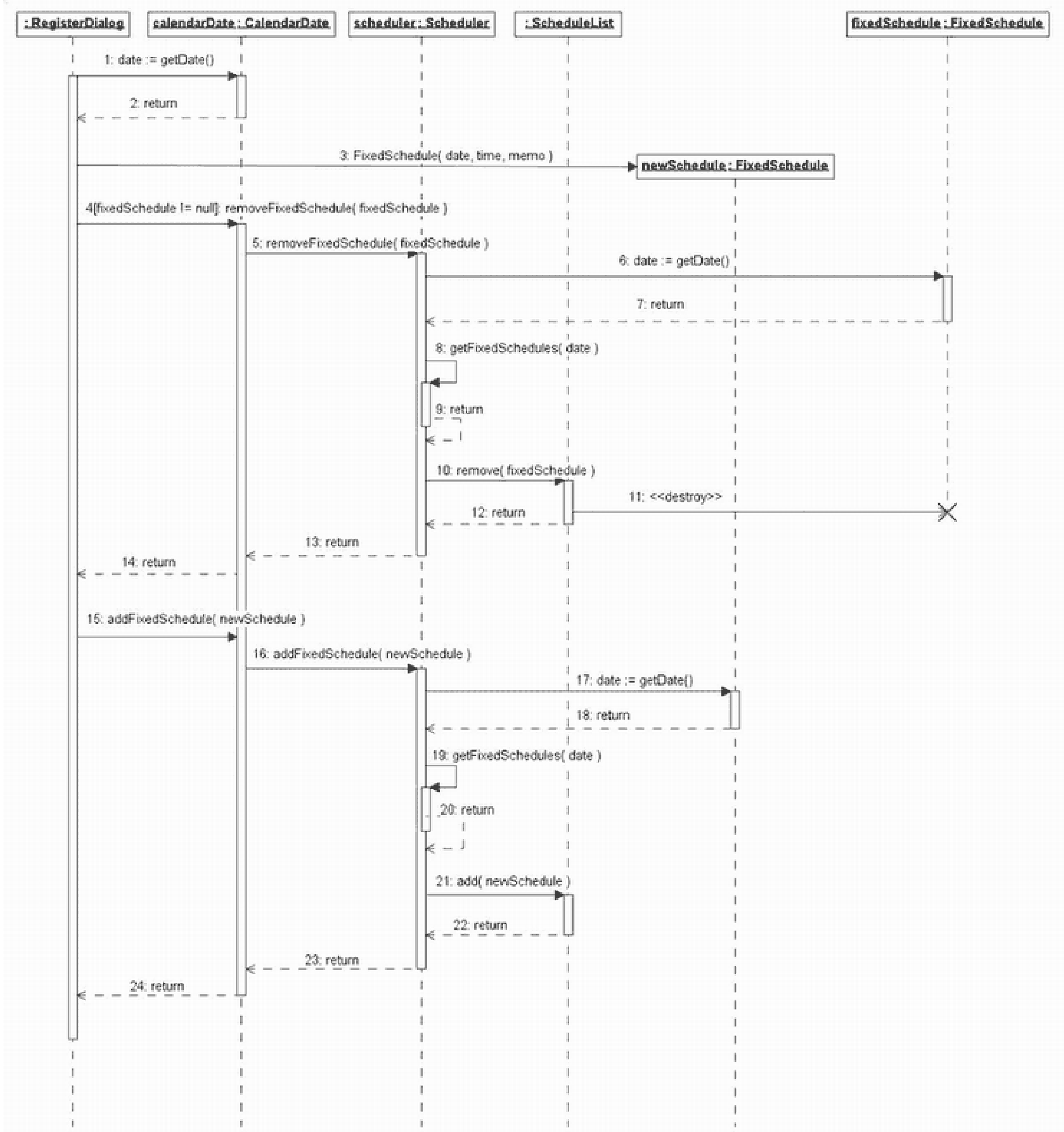

図 29: 設計シーケンス図 (scheduler)

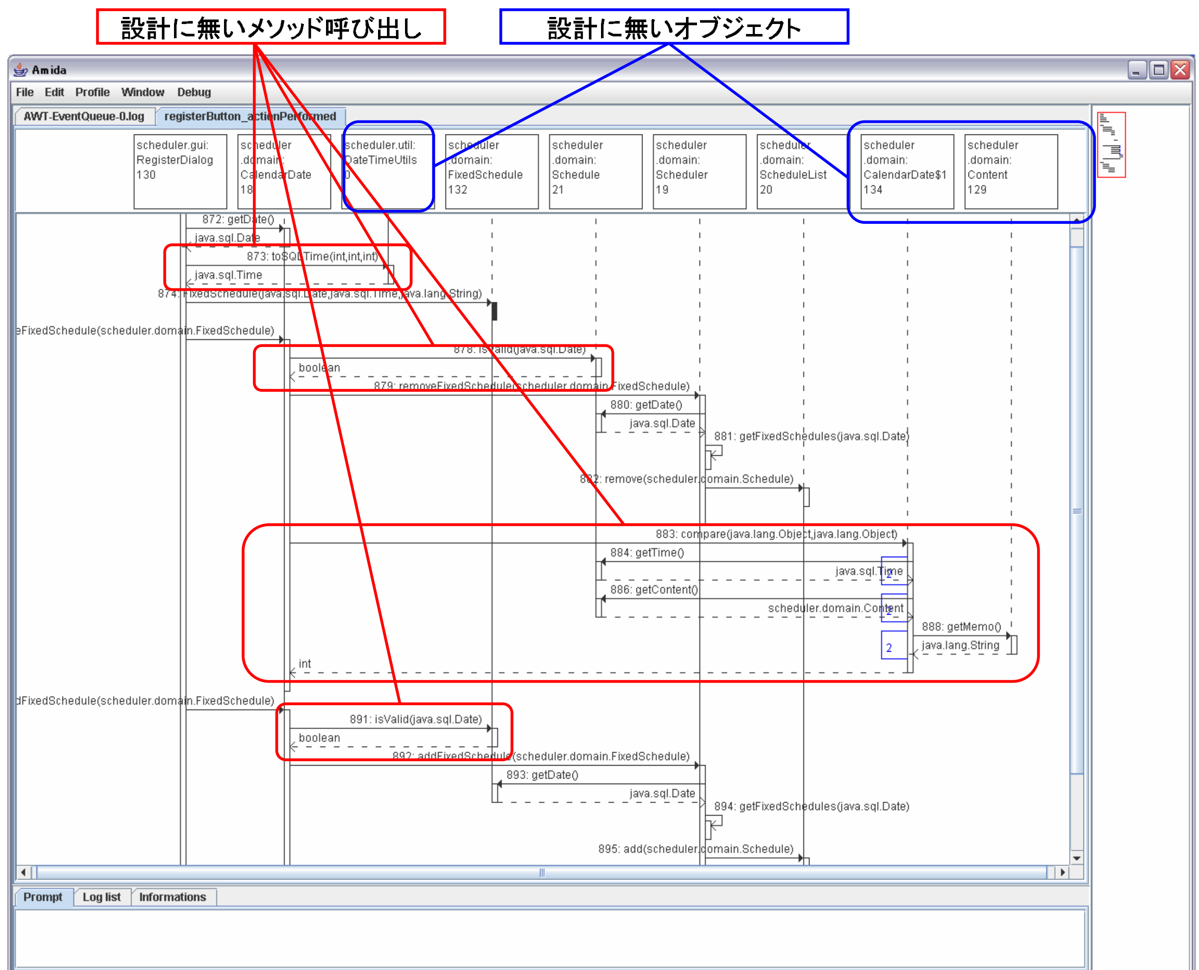

図 30: 生成したシーケンス図 (scheduler)

6 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．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 [21]．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．他には，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 [22]．

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 [3][9]．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 [5][6]．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 [9] 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 [3] 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 [17]．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 [12] 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．

より一般に，プログラムの実行時の情報から設計モデルを生成し，プログラム理解を支援する研究として，実行時の振る舞いから，そのシステムのアーキテクチャを作成する手法が提案されている [26][27]．彼らは，実行時の低レベルイベントとアーキテクチャレベルのイベントとのマッピングを行うことで，実行時の情報をより抽象度の高いアーキテクチャモデルに置き換えることに成功している．

また，Lukoit らはプログラムの特定の機能に関連する部分を表示する手法の提案を行って

いる [11]．彼らの手法では，プログラム実行中にどのコンポーネントが現在の実行と関連しているかを示す．我々は現在デバッグ環境と連動してシーケンス図を生成する研究も行っており，彼らの手法と同じように，プログラムの実行と同時に図を生成するということを行っている．また，プログラムの特定のイベントに対して，その前後の GUI 上の変化を元に，プログラムの状態の判定を行う研究もある [2].

Hamou-Lhadj と Lethbridge らは，ユーティリティーコンポーネントを検出する手法を提案している [4]．ユーティリティーコンポーネントに関するメソッド呼び出しをシーケンス図として提示する情報量を削減できれば，提示する情報量の削減に非常に効果的であると考えられ，我々の手法でも利用したいと考えている．

7　むすび

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の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げられる．

- 実行履歴を，それぞれが担当する機能ごとに分割する．
  1 つの実行履歴を複数の機能的なまとまりに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．

- ソースコードを静的解析した結果の情報を併用する．
  実行履歴の形式に着目するだけでなく，ソースコードの制御構造に沿うような形に圧縮して表示することも考えている．

- マルチスレッドのシーケンスを 1 つの図中に表現する．
  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[25] の形式を利用した表現法にする．

謝辞

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

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

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

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

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

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

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

参考文献

[1] Jon Beck and David Eichmann: Program and interface slicing for reverse engineering. Proceedings of the 15th International Conference on Software Engineering, pp.509-518, 1993.

[2] Keith Chan, Zhi Cong Leo Liang and Amir Michail: Design Recovery of Interactive Graphical Applications. Proceedings of the 25th International Conference on Software Engineering, pp.114-124, 2003.

[3] Thomas Gschwind and 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.

[4] Abdelwahab Hamou-Lhadj and Timothy C. Lethbridge: Reasoning about the Concept of Utilities. Proceedings of the ECOOP Workshop on Practical Problems of Programming in the Large, June 2004. *http://se.informatik.uni-oldenburg.de/pppl/*

[5] IBM Alphaworks: Jinsight.
http://www-6.ibm.com/jp/developerworks/java/jinsight.html

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

[7] Dean F. Jerding, John T. Stasko and Thomas Ball: Visualizing interactions in program executions. Proceedings of the 19th international conference on Software engineering, pp.360-370, 1997.

[8] Java Virtual Machine Profiler Interface.
http://java.sun.com/j2se/1.4/ja/docs/ja/guide/jvmpi/jvmpi.html

[9] 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. ソフトウェア科学会 ソフトウェア工学の基礎ワークショップ FOSE2004, pp.17-28, 2004.

[10] Moises Lejter, Scott Meyers, and Steven P. Reiss: Support for Maintaining Object-Oriented Programs. IEEE Transaction Software Engineerings. Vol.18, No.12, pp.1045-1052, December 1992.

[11] Kazimiras Lukoit, Norman Wilde, Scott Stowell and Tim Hennessey: TraceGraph: Immediate Visual Location of Software Features. International Conference on Software Maintenance, pp.33-39, October 2000.

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

[13] Michael J. Pacione: Software Visualisation for Object-Oriented Program Comprehension. Proceedings of the 26th International Conference on Software Engineering, pp.63-65, 2004.

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

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

[16] Tamar Richner and Stephane Ducasse: High-Level Views of Object-Oriented Applications from Static and Dynamic Information. Proceedings of the International Conference on Software Maintenance, pp.13-22, August 1999.

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

[18] Scott R. Tilley: Management decision support through reverse engineering technology. Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research, Vol.1, pp.319-328, 1992.

[19] 高木栄児, 山口健, 伊藤喜一, 林俊樹, 森三貴, 山内亭和: Java ではじめる UML [第 6 回]UML による設計とコンポーネント図/配置図の作成. 月刊 JavaWorld, 1 月号, 2003.

[20] The jEdit developer team:jEdit. *http://www.jedit.org/*

[21] Paolo Tonella and Alessandra Potrich: Reverse Engineering of the Interaction Diagrams from C++ Code. Proceedings of International Conference on Software Maintenance, pp.159-168, 2003.

[22] 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.

[23] Yasushi Ueda, Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue: Gemini: Maintenance Support Environment Based on Code Clone Analysis. Proceedings of Eighth IEEE Symposium on Software Metrics, pp.67-76, June 2002.

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

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

[26] Hong Yan, David Garlan, Bradley Schemerl, Jonathan Aldrich and Rick Kazman: DiscoTect:A System for Discovering Architectures from Running Systems. Proceedings of International Conference on Software Engineering, pp. 470-479, May 2004.

[27] Robert J.Walker, Gail C. Murphy, Bjorn Freeman-Benson, DarinWright, Darin Swanson, and Jeremy Isaak: Visualizing Dynamic Software System Information through High-level Models. Proceedings of Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications, pp.271-283, 1998.

[28] Norman Wilde and Ross Huitt: Maintenance Support for Object-Oriented Programs. IEEE Transactions on Software Engineering, Vol.18, No.12, pp.1038-1044, December 1992.