ソフトウェア工学 99−17
（1994．7．8）

オブジェクト指向プログラムの理解のための視覚化技法

三ッ井 欽一　中村 宏明

日本アイ・ビー・エム株式会社 東京基礎研究所

本論文では、オブジェクト指向プログラムの視覚化について議論する。プログラムの視覚化が、その挙動や設計者の意図の理解、さらには再設計や再利用の検討に有効であることは言うまでもない。問題になるのは、いかに一枚の画面に収まる程度の複雑さでかつ有効性の高い情報を抽出するかである。我々は、一般に表示するには複雑過ぎるクラス間の依存関係からより高い抽象を抽出する方法を提案する。また、クラス間の異なる種類の関係を含むグラフを、わかりやすく空間を効率的に利用して表示するレイアウト方法を提案する。

Visualization Techniques for Understanding Object-Oriented Programs

Kin'ichi Mitsui and Hiroaki Nakamura

IBM Research, Tokyo Research Laboratory

This paper discusses a problem in visualizing object-oriented programs. It is needless to say that visualization is effective for understanding the behavior and the designer's intention of a program as well as for considering the redesign or reuse of the program. The problem is how to extract information that is not only effective but also reasonably simple to show in a single screen. We propose a method to extract higher abstraction from inter-class dependency, which is generally too complicated to show; in addition, a method to layout comprehensive and space-efficient graphs that represent heterogeneous relationships between classes.

1 はじめに

本論文では、オブジェクト指向プログラムの視覚化について議論する。プログラムの視覚化が、その挙動や設計者の意図の理解、さらには再設計や再利用の検討に有効であることは言うまでもない。我々は、情報の視覚化を考える上では次の点が重要であると考える。

- 一枚の画面に収まる程度の複雑さで有効性の高い情報をいかに抽出し表示するか。
- 全体の概略図から詳細図までいかにスムーズにビューを変えられるか。

一方、プログラムを解析して得られる情報は量が膨大で複雑である。オブジェクト指向プログラム言語は、抽象化の機能を多く備えているので、プログラムを解析することにより最初に述べた目的に有効な抽象レベルの高い情報を直接抽出し視覚化することを可能にする。しかし、それでも情報量は視覚化にとって多過ぎる場合がある。従って、更に高いレベルの抽象を取り出すことを考える必要がある。高い抽象を得ることは、「一枚の画面に収める」という条件のプログラムの規模に対する限界を和らげることになるし、多くの異なる抽象レベルが扱えれば、「概略から詳細へのスムーズな移行」が可能になる。

我々は、オブジェクト間の相互作用を視覚化することに興味がある。このような相互作用を形式的に記述しようという試みは幾つかある（例えば[2]）が、現在、広く利用されている言語（例えば、C++）からはこのような情報を直接得ることはできない。直接得られるのは、「あるオブジェクトが別のオブジェクトへの参照をもつ」とか「別のオブジェクトの関数を呼び出す」といった比較的低レベルの情報であり、「幾つかのオブジェクト群が協調動作をしてあるまとまった機能を実現する」といったことは直接的にはわからない。我々は、このような言語で書かれたプログラムから、オブジェクトの相互作用としてできるだけ本質的な情報を取り出すことを試みる。

我々の視覚化システムは、プログラムから抽出できる情報を加工することでその挙動や設計者の意図を理解するのを助ける。ここで、「このような視覚化システムは、十分な設計文書が残っていれば必要ない」という主張もありえる。しかし、以下の理由でこの視覚化システムが依然として有効である。

1. プロトタイピングのような場合、厳密な設計に時間をかけず十分な文書化は行なわない場合も多い。そのような場合でも、特に共同開発時には、お互いのプログラムを理解する必要がある。
2. 現時点では、設計文書の内容と実現プログラムの間にギャップがあり、実現プログラムの説明は十分に文書化されない傾向がある。また、設計文書と実現プログラムの同期は人手によることが多い。
3. ソースプログラムは最新で、形式的で、その動作に完全に忠実な文書であり、作業者間で設計を共有するためによく用いられる。しかし、抽象度が図的表現ほど高くなく、実際に利用するのは容易でない。
4. 設計文書は、一般に量が多くなる。我々の図的表現は、そのような設計文書へのインデックスとして使える可能性がある。実際、多くのクラスライブラリのマニュアルが、継承関係の図的表現を載せているが、我々はより有効な情報をもった図を提供する。

さて、一口にプログラムの視覚化と言っても、その目的やアプローチの仕方により、その研究は多岐にわたる。Roman [8] は、プログラムの視覚化を「プログラムから図的表現への写像」と定義し、プログラムの視覚化技術を次のように分類している。

適用範囲（Scope）プログラムのどのような側面を視覚化するのか、また、視覚化の目的

抽象化 (Abstraction) どのような内容の情報を表示するか

記述方法（Specification Method）視覚化システムの側面、視覚化する情報をいかに記述するか

視覚技法（Technique）どのような図的表現を用いるか

本論文では、特に、オブジェクト指向プログラムを対象とする場合に工夫が必要になる抽象化・視覚技法について詳細に議論する。我々の適用範囲は、プログラムを解析して得られる静的な構造の視覚化である。特にクラスを持つオブジェクト指向プログラミングにおいて特徴的である、クラス間の相互作用、継承、クラスの内部構造などを視覚化する。クラスの静的解析のみで実際のオブジェクト間の相互作用を理解するには限界があるが、クラスを調べることでかなりの情報が得られる。視覚情報の記述方法については、詳細は本論文の対象外であるが、既に報告 [13] しているように、我々が開発したプログラムデータベースは、C++コンパイラの解析結果に対する Prolog 言語による問合わせのインタフェースを提供する。視覚化システムにおける抽象化や図的表現においては、数多くのパラメタを対話的に決定することが本質的なので、このようなインタプリタ言語のもつ柔軟性は非常に強力であり、重要な意味をもつ。

以降、2章では、プログラム情報の情報量の問題と抽象化の必要性、および抽象化のための具体的な方法について述べる。3章では、新しいグラフのレイアウトを提案しその有効性について議論する。

我々の視覚化技法は、C++プログラムの解析結果に基づいている。C++のような強い型付けを行なう言語からは、プログラム理解や視覚化にとって有効なプログラム設計上の情報を抽出することができる。本論文の方法は、C++ に特有なものも含まれるが、多くの議論は、一般のオブジェクト指向言語においても適用可能である。以降、説明中では、オブジェクト指向機能に関してC++における用語を用いる。

2 プログラム情報の抽象化

2.1 従来の技術

我々は、オブジェクト指向プログラムの特徴的な側面、特にオブジェクト間の相互作用を視覚化したい。容易に思いつく方法は、静的解析の結果得られるクラス間の様々な関係をグラフ化することであるが、既存のクラスブラウザ [1][7] 等をみると、クラスの継承関係程度のサポートしかない。関数呼出しのグラフを表示できるものもあるが、これをクラス間の関係として理解するのは容易ではない。これらのブラウザがたまたま継承以外の関係をサポートしていないのではなく、本質的に困難な問題があると考えるべきである。

問題なのは、クラス間の一般的な関係がサイクルを含むグラフであり、また、ほとんどのクラスどうしが何らかの関係をもっていてグラフが大きくなってしまうので、結果として非常に複雑なグラフになってしまうことである。一方、継承関係はこの大きなグラフのかなり小さなサブグラフであり、かつ、十分意味があるので、事実上唯一用いられているのである。

表示技術的には、グラフがかなり大きくても適当なレイアウトアルゴリズムを用いて実用的な速度で表示することは何の問題もない。しかし、そのグラフから何かを読みとることは、実際には困難な場合が多い。結局、極端に局所的なグラフに限定して表示し、そのような情報の断片を追っていくという形になる。これでは、図的表現の効果が十分発揮されているとは言えない。

2.2 抽象化の必要性

ここでは、実際のアプリケーションプログラムからどの程度のプログラム情報が得られるかを示す。我々のプログラムデータベースは次の情報を含む。

- プログラム中で宣言されたシンボルの属性（名前、型等）
- プログラム構造（クラス-メンバー構造など宣言どうしの入れ子構造と継承関係）
- シンボルのクロスレファレンス

	#Lines	#Classes	#Inh	#F-Calls	#C-Calls
Unidraw	38719 (53501)	245 (436)	394	10506	2173
Pv	36817 (120042)	242 (1399)	667	15996	2662

図 1: プログラムデータベース内の情報量

この情報から、継承階層、関数の呼出し関係、クラス間の依存関係などが導出できる。図 1は、これらの情報量の多さを示している。アプリケ

ーションプログラムの例としては、Unidraw[11] (InterViews[5] 上に作られたエディタ作成のためのフレームワーク) と Pv (我々が開発しているプログラミング環境 [6] のユーザインタフェース部) を取り上げた。図 1は、アプリケーションの規模として、行数、クラス数を、グラフ表示の対象としたい関係として、順番に継承、関数呼出し、クラス間の依存関係の数を示す。(クラス間の依存関係としては、ここでは、メンバー関数どうしの呼出し関係をクラス間に抽象化したものを用いた。) なお、行数、クラス数は、上段は利用するクラスライブラリのインタフェースを含まず、括弧内は、それを含む。継承、関数呼出し、依存関係はライブラリへの参照を含む。Unidraw, Pv とも現実的な中規模のアプリケーションと言える。図 1は、クラスの依存関係などをグラフ表示するには、さらに抽象化や簡約化が必要であることを示している。

2.3 クラスの相互依存に基づいた抽象化

2.2の結果を鑑みると、関数呼出しのような情報をクラス間の関係に縮退したとしても、(このようなツールが威力を発揮するような) 適当なサイズのアプリケーションに対しては、図的表現の複雑さが許容範囲を越えているといえよう。クラスよりも高い抽象をプログラム情報から抽出できないであろうか。我々は、クラスの集合の中にお互いに高い依存度をもったグループがあると考え、それらを抽出することを考えた。このグループを島と呼ぶことにする。島は、クラス間の依存関係を適当に定義 (例えば 2.2で用いた関数呼出し関係) し、その依存関係を有向グラフとみたとき、その強連結成分を 1 点に縮約した凝縮グラフを求めることにより得られる。すなわち、2 クラス間に相互に依存関係のパスがあればそれらは一つの島に縮約される。

2 島間でそれぞれに属するクラス間に依存関係がある場合、島の間にも同方向の依存関係があるとする。すると島の間の関係はサイクルを含まない有向グラフになる。例えば、図 2のようになる。A から G はクラス、矢印は依存関係を表しており、4 つの島ができている。

島が意味するところは、「ある島に属するクラス群が協調動作をすることにより別の島に対してあるサービス (群) を提供する」である。このような抽象化を利用すると、島の間の関係により大雑把なプログラムの構成を把握し、次の段階で個々の島についてその内容を理解するという手順が可能になる。

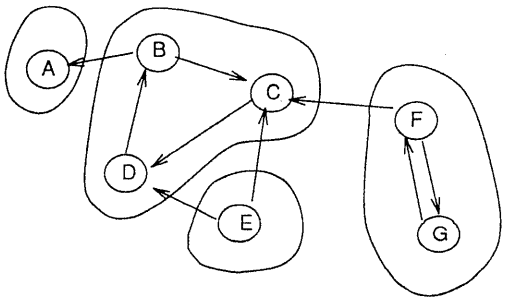

図 2: クラスの依存関係と島

この島に相当する情報は、従来のプログラミングでは、ソースコードが置かれるファイルの場所やシンボル名などに陰に表現されている場合が少なくないが、我々の方法はこのようなプログラミングコンベンションを全く仮定しない。

2.4 我々の方法の効果と改良

さて、ここで当然湧く疑問は、このような島が現実のプログラムでどのくらいの粒度で存在するかということである。我々の実験では、予想以上に大きな島または小さな島ができてしまう傾向があることがわかった。図 3に前述の Unidraw と Pv に対して、その結果を示す。表は、島の大きさごとの分布を表している。Unidraw では、大きな単一の島ができた。Pv では、後で述べる方法により島の分解を行なって適当な大きさの島が得られている。

	#Classes	#Islands	#Calls
Unidraw	156	1	1019
	2	3	3 - 4
	1	238	0
Pv	40	1	190
	9 - 14	3	29 - 58
	2 - 3	3	4 - 5
	1	162	0

図 3: 抽出された島

我々は、その原因を調べるために、図 4(a) のようなパターンに注目した。細い矢印は一般の依存

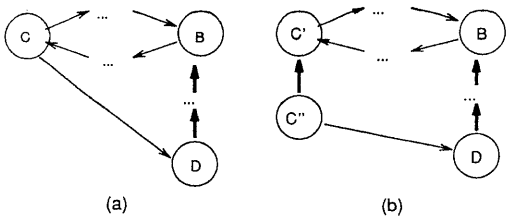

図 4: (a) 異なる抽象度のクラスが分離できない場合 (b) クラスの分離

関係、太い矢印は依存関係としての継承関係を表している。このパターンの意味するところは、「あるクラスCがある親クラスBと相互依存関係にあり、かつ、Bの導出クラスDにたいしても依存関係がある」である。このパターンは、「理想的には、クラスCとBが協調動作するならば、それらはより継承階層が下位のクラスに依存すべきではない」という仮説に基づいている。このようなパターンは、注意深く設計されたプログラムにおいても存在することはあるが、次のような場合は注目に値する。

- クラスCが継承階層を持たない。または、継承階層の比較的上部にある。
- クラスDが継承階層の比較的下部にある。

このようなパターンを発見し、検討した結果、図4(b)のように変更することが可能であれば、より多くの島が抽出できる可能性がある。実際、Pvの例において、１０個ほど問題のある依存関係を発見し、図3を得た。このことは、視覚化システムの利用者がこのようなパターンを発見し、分割をするためのヒントを与えないと我々の方法はうまく機能しないことを示している。しかし、別の見方をすると、この例の場合プログラムの設計が我々の方法と相性が悪かったのであるが、設計者が図4のような変換を行なうことにより、いわゆるFramework[3]のような高い抽象を積極的に継承階層に反映させるように努力すればよかったわけである。これは、「複数の直接の導出クラスがなければできるだけクラスを融合する」といったプログラミングスタイルとは全く異なる。このことは、プログラムの理解にも有用であり、さらに、クラス間の不要な依存関係を減らし、(特にC++プログラミングでは顕著であるが) プログラム開発の効率を高めるのにも役立つ。

更に、小さな島ができてしまう問題も存在する。これは、クラスのインタフェースと実現を別のクラスとするプログラミング技法が用いられた場合である。実際、InterViewsにおいて多用されている。図5でこの状況を説明する。いま、A、Bがインタフェースの定義のみ行なう親クラスで、C、Dがそれぞれの実現となる導出クラスであるとする。太い矢印は継承、細い矢印は関数呼出しに対応する。導出クラスどうしは直接呼出し関係を持たず、インタフェースにある仮想関数を経由する。意味的には親クラスどうし（または導出クラスどうしに）に依存関係が期待されるにもかかわらず我々のアルゴリズムでは島を検知することはできない。結果として、小さな島が多くできてしまう。

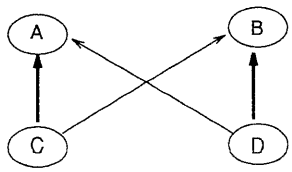

図 5: インタフェースと実現が異なるクラスの場合

これに対処するには、あるインタフェースクラスで宣言された全ての仮想関数に注目し、実現クラスのそれらの仮想関数の定義から得られた他のクラスとの関係が、インタフェースクラスにおいても成り立っていると考えればよい。実現クラスが複数ある場合には、実現クラス間での共通部分、または、和をとればよい。共通部分をとった場合、異なる実現間に一般性のある情報が得られ、和をとった場合、実行時にはあり得ない依存関係を含んでしまう可能性があるが、これを静的に判定するのは一般には困難である。

ここで、もう少し仮想関数の扱いを一般化してみる。あるクラスが仮想関数を含む場合、次のような選択がありうる。

1. 仮想関数の中身を全く考慮しない。
2. そのクラスでの中身および導出クラスでの中身の共通部分を考慮する。
3. そのクラスでの中身および導出クラスでの中身の和を考慮する。

1は上で議論した技法が用いられている場合全く情報がなくなってしまうが、一般の状況では、「導

出クラスにおけるいかなる再定義に対しても不変な情報」である。クラスの設計者がこの観点から積極的に仮想関数と一般関数の使い分けを行なっている場合には意味のある選択である。2、3は、上で議論ようは技法が使われていることがわかっている場合にはその意味はわかりやすい。より一般には、曖昧さを増加させたり、再び大きな島をつくってしまうかもしれない。

2.5 フィルタリングと強調

これまで、議論したような技法を用いたとしても、プログラムのグラフ表現が十分理解しやすい程度に一画面に収まるという保証はない。そのような場合、視点や目的をより詳細化し、更にグラフを簡約したり、注目すべき部分を強調する必要がある。

例えば、2.3では、クラス間の依存関係を厳密に定義しなかったが、以下のようなものが考えられる。

1. 継承関係
2. 大域変数、メンバー変数、関数引数、局所変数、関数の戻り値を経由した別のクラスへの参照
3. メンバー関数の呼出し関係 （2の部分集合になっている）

依存関係として何を採用するかにより、視点や視覚化の目的は異なり、また、情報の複雑さも変化する。

より自明でない例として、我々は[12]において、プログラムを視覚化する上で、プログラムから得られる情報を直接表示した場合、本質的ではない情報が多く表示されるという問題を指摘した。例えばコレクションクラスのように、上位設計では関連として扱われるが、実現言語が関連をサポートしていないため実装上導入されるものは、必要以上に図的表現を複雑にしている可能性がある。上のような依存関係を検討することにより、このようなクラスを発見できる可能性がある。例えば、２つのクラスが2の依存関係は持つが、3の依存関係は持たないとする。これは、コレクションクラス等ではよくみられるパターンである。このようなクラスが発見されたら、これをグラフの点なく線で表現する。

3 視覚技法における工夫

2章では、クラス間の関係に関して抽象化を行ない、いかにそのグラフ表現を簡約化できるかについて議論した。ここでは、そのようにして得られたグラフを、できるかぎり画面を有効に利用して表示するための技術について述べる。

3.1 関連する研究と問題点

オブジェクト指向プログラムの視覚化においては、異なる種類の情報を同一グラフ中に表示したいことが多い。特に、継承関係以外のクラス間の関係（例えば、Has-a 関係）を表示したい場合、それらの関係は、親クラスから導出クラスに継承されると考えるのが自然な場合が多い。これを自然に表現する方法の一つは、対象となる関係と継承関係を同一のグラフに含めることである。

グラフの自動レイアウトに関しては、サイクルを含むグラフが扱えるものを含めて多くの研究がなされている[10]。しかし、上で述べたような異種関係がある場合にそれらが必ずしも効果的でない。我々の実験では、単一の関係には有効であるレイアウトを適用し、辺の色や線種を関係の種類ごとに変えたとしても理解しやすいグラフは得られないことが多い。そこで、我々は、それぞれの関係にたいして異なるレイアウト基準を用いることを考えた。これを次に述べる。我々の方法に似ているものとして、[9]のような入れ子グラフを用いるものがある。我々の方法も一種の入れ子グラフとみることもできるが、実際の計算方法は異なる。

3.2 異種関係の表示レイアウト

ここでは、我々の具体的な目的は、クラス間での継承関係とそれ以外の関係を同時に表示することであるとする。継承関係は、サイクルを含まない有向グラフであり、それ以外の関係は、サイクルを含む有向グラフとする。ここでは説明の都合上このような具体的な問題設定をするが、ここで述べるレイアウトアルゴリズムは、一般に半順序関係と任意の関係を同時にグラフ表示したい場合に用いることができる。

このレイアウトの目標は、図6のように同一の継承階層に含まれるクラスは、同心円状に、中心にい

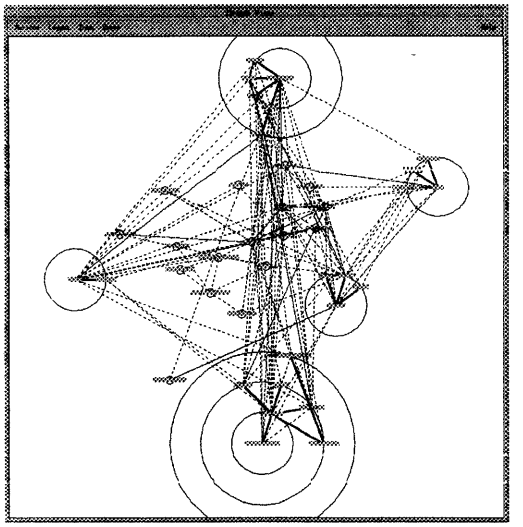

図 6: 継承と呼び出し関係の同時表示

くほど上位クラスになるように配置され、そのような複数の継承階層は、表示画面上でできるだけ分散配置されるようにすることである。クラスはグラフ中の点として表現され、継承およびその他の関係は、色または線種を変えた辺によって表示される。

紙面の関係上、アルゴリズムの詳細は述べないが、基本的な方針は次のとおりである。まず、クラスの集合を継承関係の連結度で分類する。ここで分類された個々の継承階層をコンポーネントと呼ぶことにする。2コンポーネント間の連結度は、それぞれのコンポーネントに含まれるクラスどうしに継承以外の関係が少なくとも一つあればコンポーネント間に同様の関係があるとする。コンポーネントが適当に分散するように「バネ」アルゴリズム[4][10]を用いる。「バネ」アルゴリズムは、グラフのノード間に適当なバネをはったとき物理的に均衡する状態を非線形最小化問題として解くことでグラフのレイアウトを行なうものである。バネの長さや強さは、各コンポーネントの大きさやコンポーネント間の連結度の強さなどを考慮して決める。コンポーネントの配置が決まったら、クラスの配置を決める。このとき「バネ」アルゴリズムをクラス間に適用する。このとき、「クラスが継承の深さに応じた同心円上のみ動く」という制約を与える。更に、これだけの制約では一般に同心円上のクラスが集積しすぎる傾向があるので、これ以外に幾つか制約を与えて最適な位置の探索を行なう。

図6は、実際にPvから抽出された島のひとつをこのアルゴリズムを用いて表示したものである。

ここで、例えばSmalltalkのクラス階層のように、全てのクラスが単一または小数の親クラスを持ち、グラフが非常に少ないコンポーネントしかもたない場合があることに注意する。このような場合は、ある程度、抽象度の高いクラスを取り除くことで、幾つかのコンポーネントに分解する。

「バネ」アルゴリズムの利点の一つは、空間を有効利用できることである。また、グラフ上の幾つかの点を固定するといった制約を、アルゴリズムの性質上自然に与えられる。欠点は、計算に多くの時間を必要とすることである。

3.3 同心円レイアウトの応用

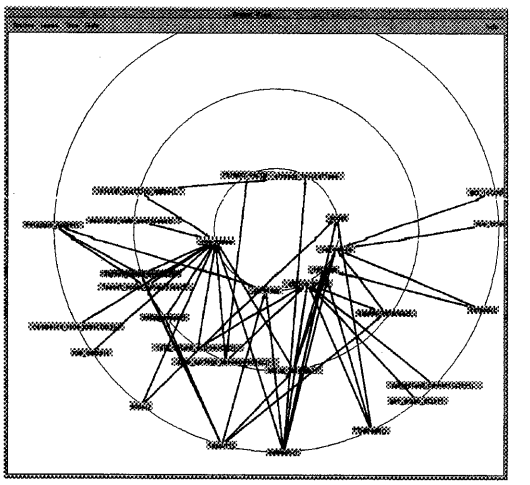

図 7: メンバー変数とメンバー関数の関係

3.2の議論では、継承関係を中心に考えたが、別の関係に応用することもできる。図7は、あるクラス内のpublicメンバー関数、privateメンバー関数、メンバー変数をそれぞれ異なる同心円に配置し、オブジェクトが外からの呼出し（または事象）によって状態を変化させる様子を表現している。また、この表現は、オブジェクト内の構造を直観的に表現しており、クラスの分解等を検討する上で有益である。なぜなら、「バネ」がメンバー間の依存度に基づいたクラスタリングを行なうからである。

オブジェクトの内部構造のみを表示する場合、3.2のような異種関係を考える必要はないが、従来のレイアウトアルゴリズムを用いるよりも我々の同心円を用いるもののほうが、空間を効率的に利用している。

4 結論

我々は、クラス間の依存関係を視覚化する際に、その情報量が多過ぎる問題点に着目し、グラフを簡約化するための手法を提案した。また、その手法がプログラムの設計によっては、うまく機能しない場合があることを示し、この場合の対処の方法についても言及した。

更に、視覚化において、異種関係を同時に表示する必要性を指摘し、これを、従来の方法よりも空間を有効に利用してわかりやすく表示するためのレイアウト方法を提案した。

視覚化においては、最終的に一画面に収まる程度の情報量になるように、いかに抽象化し図的表現を簡約化するかが問題になる。この過程をどのような対象プログラムに対しても自動化することは非常に困難である。したがって、この問題は本質的にユーザとの対話によって解かれなければならない。そのためには、システムができる限り多くの支援機能をユーザに対して提供すべきである。我々は、そのような機能を調べあげるために、実際のアプリケーションを対象とした実験を行なっていく必要があると考える。我々のプログラムデータベースはそのための重要な材料を提供する。

また、今後の別の可能性として、実行時の動的な情報の利用を考えている。動的な情報もその量は非常に多い。例えば、我々の実験では、たとえ小さなサンプルプログラムでもその実行トレース情報から得た関数呼出しのグラフはかなり大きなものになる場合がある。この場合、本論文での方法と組み合わせて、静的な情報によりある程度視点を絞った後、実行トレース情報を重ね合わせたり、アニメーション等を用いて時間情報の表現を加えることにより、プログラム理解に高い効果をあげることができると予想される。また、静的解析だけでは限界のあったオブジェクトのインスタンス間の相互作用や仮想関数に関する曖昧さの解消などを扱うことが可能になる。

参考文献

[1] *AIX XL C++ Compiler/6000, Source Code Browser - User's Guide*, IBM Corporation, 1992.

[2] Helm, A. R., Holland, I. M., and Gangopadhyay, D.: Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, *Proc. of OOPSLA'90*, 1990.

[3] Johnson, R.E. and Foote, B.: Designing Reusable Classes, *Journal of Object-Oriented Programming*, Vol. 1, No. 2, 1988.

[4] Kamada, T. and Kawai, S.: An Algorithm for Drawing General Undirected Graphs, *Information Processing Letters 31*, 1989.

[5] Linton, M. A., Calder, P. R., Interante, J. A., and Vlissides, J. M.: *InterViews Reference Manual Version 3.1*, Stanford University, 1992.

[6] Mitsui, K., Nakamura, H., Law, T. C., and Javey, S.: Design of an Integrated and Extensible C++ Programming Environment, *Object Technology for Advanced Software*, LNCS 742, Springer-Verlag, 1993.

[7] *ObjectCenter - User's Guide*, CenterLine Software, Inc., 1993.

[8] Roman, G. C. and Cox, K. C.: Program Visualization: The Art of Mapping Programs to Pictures, *Proc. of ICSE 14*, 1992.

[9] Sugiyama, K. and Misue, K.: Visualization of Structural Information: Automatic Drawing of Compound Digraphs, *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 21, No. 4, 1991.

[10] Tamassia, R., Battista, G. and Batini, C.: Automatic Graph Drawing and Readability of Diagrams, *IEEE Transactions on Systems Man, and Cybernetics*, Vol. 18, No. 1, 1988.

[11] Vlissides, J. M. and Linton, M. A.: Unidraw: A Framework for Building Domain-Specific Graphical Editors, *ACM Trans. of Information Systems*, Vol. 8, No. 3., 1990.

[12] 中村, 安田, 大平, 三ッ井: オブジェクト指向ソフトウェア開発におけるプログラム理解支援, 情報処理学会研究報告, 94-PRG-15, 15-4, 1994.

[13] 中村, 安田, 三ッ井, S.Javey: 拡張可能なC++ソースコード・ブラウザ - プログラム・データベース, 情報処理学会全国大会, 7Q-06, 1992.