Extendable Object Visualisation for Software Reengineering

Jörg R. Mühlbacher, Peter R. Dietmüller, Markus Jöbstl
Department of Information Processing and Microprocessor Techniques
Johannes Kepler University of Linz
Altenbergerstraße 69
A-4040 Linz, Urfahr

Abstract

This paper describes a reengineering tool which assists the process of understanding the functionality of unknown software, particularly object oriented programs. In contrast to many other tools which analyse the source code, the inspected program is analysed at runtime by a concurrent process running in parallel. Information on all objects allocated by the inspected program is collected, in particular the dynamic type of each inspected object is determined. In contrast to the static type, the dynamic type of an object can only be determined at runtime. Each object is visualised by a corresponding visualisation class. Visualisation classes for well known data structures like binary trees can be used from the beginning. New visualisation can be derived by class extensions or can be added simply. The inspected program can be halted at specific locations to update the visualisation. Also, updating the visualisation can be triggered by specifying watch points.

Introduction

Modifying programs, even somebody else's software is a frequent task of software engineers. Due to bugs or changing requirements software has to be updated regularly. Changing software requires exact knowledge about the function of the particular piece of code and how it interacts with other parts of the whole system.

If you have written it by yourself you can rely on your knowledge of the software acquired during the development process. If, however, the author can't be reached, if he has left the company for instance, you only can rely on the software documentation as an information source. Since writing documentation isn't normally the highest priority task, especially when the project is late, it could become a difficult job to familiarise yourself with how the subject code functions. This can become even worse for object oriented programs, because the interaction between objects is difficult to comprehend.

We tried to overcome this situation by providing a reengineering tool which helps to analyse object oriented programs during runtime. In contrast to other tools which analyse the source code, our tool watches what is happening during the execution of the analysed program. Since the most important parts of an object oriented program are the objects and their interactions we are focusing on objects. Object oriented programs typically create many objects during runtime. Our tool gathers information about all objects currently existing and determines the type of each object. Since the current type of an object cannot be determined by analysing the source code only it is clear that the type of an object can be determined at runtime only.

Simply knowing the type of an object isn't very useful. Instead we want to visualise the objects according to their type. The idea was that an object of type CAR should be drawn as a car if it's suitable. A more technical example would be a simple data structure such as an array. This could be shown as in figure 1. So we want to draw data structures, too. Not only data structures as simple as arrays, but also dynamic data structures such as linear lists, trees and graphs.

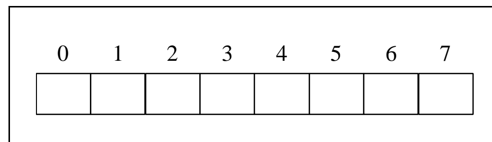

Figure 1: Visualisation of an array

Many people have thought about the best visualisation for specific data structures. Good examples can be found in [Brown91], [Stasko90] and [DingMateti90]. However we think that the question for the best visualisation can't be answered simply. The answer depends on the situation in which the data structures are used. Consider the following example of a traffic simulation. A queue of cars waiting in front of a gas station could be visualised as a queue of cars. If, however, it's interesting how many cars are waiting it could be enough to show the number of waiting cars instead of drawing the whole queue. In other situations each car could be of interest and then it would be better to draw each car. Furthermore it could be interesting how long a car has been waiting. It could be useful to draw a red car if the corresponding car has been waiting longer than a specified amount of time.

This little example should show that the way an object is best visualised depends on the situation in which the object is used. In other words: the appropriate visualisation of an object is context sensitive. The context (also: application area or domain) brings in semantics associated with the object to be drawn.

This leads to the idea of having more than just one visualisation for an object type in order to provide different visualisations for such an object type. On the other hand it is unknown a priori when and within which domain an object will be used and since we can't provide visualisations for all conceivable application domains, we have to make the system flexible in such a way that everybody can extend it by adding new visualisations for specific object types. In order to be able to use the tool from the beginning, a set of standard visualisations is provided which can be used unchanged or may be extended according to specific applications.

A specific purpose of this tool is to assist the process of learning the exact functionality of an unknown piece of software, particularly object oriented software. Since the interaction and relationships between objects are an important part of the functionality of object oriented programs we focus on drawing (visualising) objects. Using proper graphical representations for objects should help to speed up the process of understanding unknown programs.

We have to overcome the following situation: if the visualisation routines must be built into the unknown program it has to be analysed in advance of the visualisation to know at which points the visualisation routines should be called. At the time one is visualising the unknown program it is no longer unknown, because it has already been analysed to some extent and the main structure is revealed. In this case the benefit of the visualisation would not be significant because one has analysed the program already. Nota bene: this approach is only possible if the source code is available!

Another argument is that changing the program could change the functionality of the program. It's clear that the visualisation extends the program's functionality but it could change its primary function in such a way that it doesn't work the same way as before visualisation routines were added. This could happen if the visualisation routines aren't called correctly, for instance. This would be an inconvenient situation because you wouldn't learn the original functionality of the program .

Therefore changing the unknown program by adding “draw commands” could be counterproductive. That's why it was an important goal that there should be no need to change the source code of the visualised program. The visualisation tool should visualise programs running in parallel without any changes to the inspected programs.

The most important aspect and consequence is: there is no need to have access to the source code! Knowledge of the specification of the classes would be helpful, but principally even this information is not necessary.

The presentation of the above ideas are divided into three parts. The first chapter describes the basic concepts. Then a prototype is presented and various implementation details are described. Finally a case study is presented.

Concept

Figure 2 shows the overall structure of our tool. The left part shows two different programs called "Program 1" and "Program 2". Both of them are to be visualised. As shown they use a runtime system. Besides many other responsibilities it manages the allocation and deallocation of dynamic memory. Object oriented programs normally use dynamic memory intensively for allocating objects.

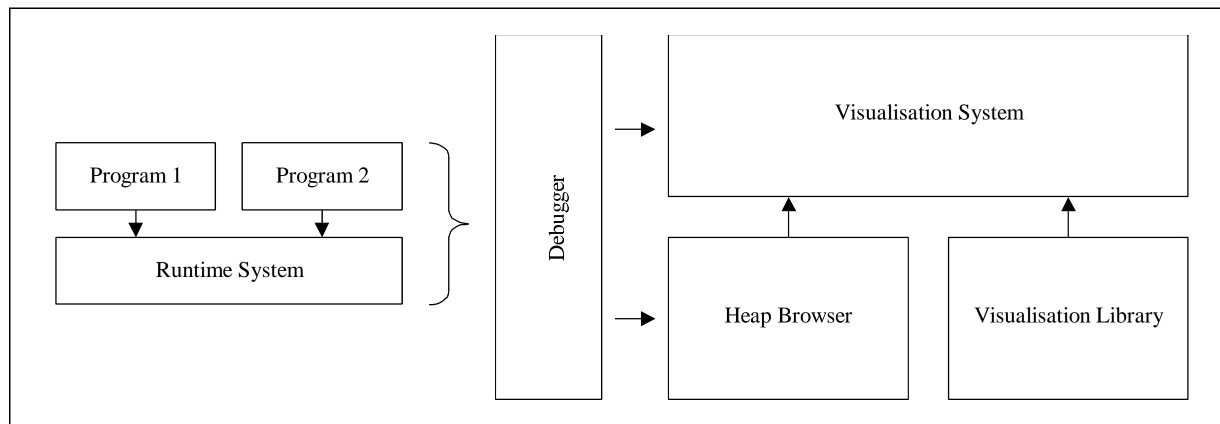

Figure 2: Structure of the visualisation system

The next component is a debugging component which gains access to the memory of the subject parallel processes and can halt the programs at specific addresses. Primarily the heap browser is responsible for selecting objects for visualisation and for determining the type of these objects. The visualisation library provides a set of visualisation classes which can be used unchanged or extended according to special needs. The visualisation system integrates all parts and provides the user interface.

Heap Browser

The part of the runtime system which manages dynamic memory is often called heap management. Consequently we name the component which is responsible for scanning the dynamic memory the “heap browser”. Since modern operating systems deny access to the memory of other processes, a debugging component is used which gains access to that memory.

The heap browser is responsible for three specific tasks in particular:

- It scans the heap of other processes in order to present to the user all objects currently allocated by the inspected programs.
- It determines the type of each object and the names of their fields.
- It lets the user select any number of objects and sends the addresses of the selected objects to the visualisation system.

Visualisation library

The visualisation library provides visualisation classes which can be used to create visualisation objects. Visualisation objects are very important because they are responsible for drawing the visualisation of an object. As one will see later there is a basic visualisation object for each object that is visualised.

We have to distinguish between “visualisation objects” and “visualised objects”.

A visualised object is an object existing within the inspected program. It is an instance of a class defined in the original (inspected) program.

A visualisation object is an object belonging to the visualisation system. Its purpose is to provide any information which allows the system to draw the associated visualised object. Each visualisation object holds a reference to a visualised object enabling the system to draw exactly this object. (see fig. 3)

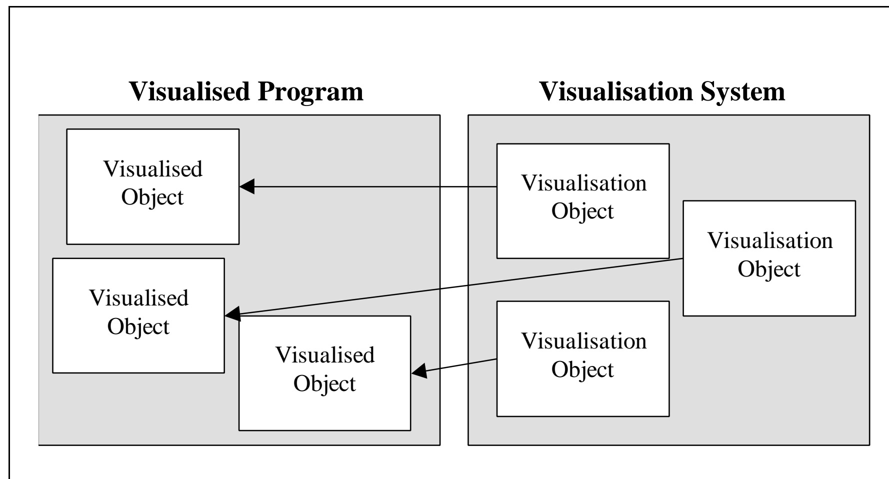

Figure 3: Connection between visualisation object and visualised object

If there were only one “visualisation object” class, there could be only one type of visualisation for all types of visualised objects, or the visualisation object must implement visualisations for all possible object types. Neither way is practical. In the first situation every object would be shown in the same way and the user wouldn't gain any further information from the graphical representation. The second way would mean that the class would have to be changed every time a new type of visualised object is added. Therefore we decided to build a set of classes organised in a class hierarchy.

Each visualisation class is responsible for exactly one type of visualised object. For instance there is one visualisation class for singly linked lists and one for binary trees. The visualisation system must know which visualisation class is responsible for which type of object. Therefore there is a simple database which links a visualisation class to an object type. In our example this database contains the information that the class VLinkedList is responsible for visualising objects of type LinkedList.

The visualisation classes are organised in a class hierarchy. This means that all visualisation classes are inherited from a base visualisation class. Since all visualisation classes are direct or indirect subclasses of this base class they inherit all methods and functions of the base class. The base class defines therefore the minimum interface which every visualisation class provides.

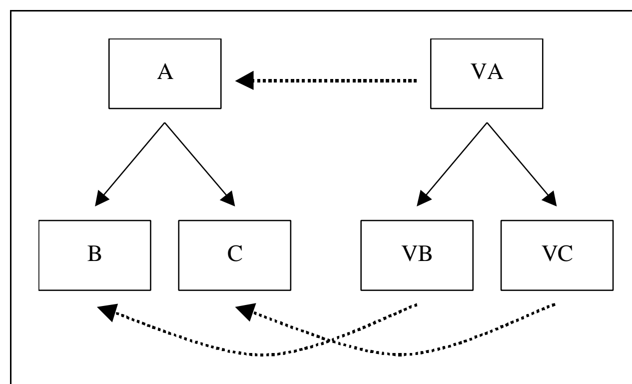

Figure 4: Class hierarchies of visualisation classes and visualised classes

We pointed out that a visualisation class is responsible for exactly one type of object. Figure 4 shows an example where the visualisation class VA is responsible for visualising objects of type A. Class A

has two subclasses named B and C. These have their own visualisation classes named VB and VC. In addition to class VB objects of class B can be visualised by the superclass of VB, namely VA. This leads to the following observation: an object can be visualised by all instances of its specialised visualisation classes and by all superclasses of the specialised visualisation classes. Consequently objects of class B can be visualised by means of the specialised class VB and all superclasses of VB, for instance VA.

This leads to an easy extension mechanism of the visualisation library. Every time a new object type is added it can be visualised either by an existing visualisation class or by a new visualisation class. First a new entry to the database must be added, which links the new visualisation class to the new object type. But if the new object type is a subclass of an already existing class the creation of a new visualisation class can be omitted. The new object type can be visualised by all visualisation classes responsible for visualising any direct or indirect superclass of the new object type. If we add a new object type D as subclass of C to our example (see figure 5), all visualisation classes for C and A can visualise the new object type D.

In this situation we don't need to add a new entry to the database. But it could be possible that the visualisation class of the superclasses can't visualise the new object type in the right way. Since they don't know anything about the new object type they can't show any new functionality. In this case we can add a new visualisation class VD which is responsible for representing objects of type D. For that we need a new entry to the visualisation database.

It is convenient to make VD a subclass of VC because it is very likely that the visualisation of object type D is similar to that of type C. Then the visualisation class VD can inherit the visualisation of VC and adapt it to the new object type D. However it is not necessary that the visualisation library builds up the same class hierarchy. It depends on the required visual representation.

Error! Objects cannot be created from editing field codes.

Figure 5: Addition of visualisation class VD

This little example shows that extending the visualisation library is an easy task. If an existing visualisation class already provides what you need, you just link it to your new object type by adding an entry to the visualisation database. If you want to implement a new graphical representation you are free to write a new visualisation class which normally inherits many features of an existing visualisation class.

In order to make integration of new visualisation classes easy we decided to put all visualisation classes into modules, which are loaded at runtime as dynamic link libraries (DLLs) or shared libraries. The visualisation system is responsible for loading the correct module at runtime. It gets the information about which class is in which module from the visualisation database. So each entry of the visualisation database consists of the three parts: the name of the object type, the name of the visualisation class and the name of the dynamically loadable module.

Visualisation System

Provided with this information an object will be visualised the following way: First the visualisation system determines the type name of the object. Then it looks for entries in the visualisation database which match with the object type's name. This search provides a number of visualisation classes and their load modules which can be used to visualise the specified object. The visualisation selects the first one, which is the default visualisation class, and loads the module containing the visualisation class. After loading it creates an instance of the visualisation class. The created visualisation object is initialised and gets the address of the visualised object from the visualisation system. From now on the visualisation object is responsible for getting the current state of the visualised object and for drawing the object within the visualisation system. So the visualisation works with a number of visualisation objects each responsible for exactly one visualised object (see fig. 3).

Up to this point the visualisation system displays the state of an object at one specific time. However, it's interesting to see when objects change their state or when objects are created or destroyed, which can't be achieved by showing a single snapshot. So we need a mechanism for updating the visualisation and for stopping the inspected program in order to be able to grasp what is shown by the visualisation.

Updating the visualisation is delegated to the visualisation objects. Every time the visualisation should be updated the visualisation system sends a Paint message to all visualisation objects. They determine the current state of their visualised objects and draw the new state.

To trigger off such update events we need debugging functions such as halting a program at a specific address. In conjunction with source level debugging information we show the source code to the user and let him decide when to stop the program and when to update the visualisation.

To improve the usability we provide different types of breakpoints which are shown in Table 1. The first type corresponds to common break points which halt a program at a specific location and update the visualisation afterwards. The inspected program isn't continued until the user restarts the program. The second consists of break points which halt the inspected program just for updating. The inspected program will be resumed automatically. We called this type " watch points".

Designation	Triggered through	Updates visualisation	Interrupts execution
Code break point	Reaching a location in the code	yes	Yes
data break point	Change of specific data	yes	Yes
Code-data break point	Reaching a location in the code of a method called for a specific instance	yes	Yes
user break	User	yes	Yes
Code watch point	Reaching a location in the code	yes	No
data watch point	Change of specific data	yes	No
Code-data watch point	Reaching a location in the code of a method called for a specific instance	yes	No

Table 1: Break Points and Watch Points

If the user has selected a data structure which consists of more than one object, such as a binary tree for instance, the visualisation system creates a visualisation object for the whole data structure and a visualisation object for each object of the data structure. Furthermore for each reference between two objects a special reference visualisation object is created, which is responsible for visualising the reference. Therefore the graphical representation of a reference can be changed by implementing a new visualisation class.

The visualisation object for a data structure is also responsible for checking whether all objects found during a previous update process are still existing and if new objects have been added to the structure. The existence check is done by each visualisation object itself. It checks if the address of the visualised object is still valid. If it's valid, the object still exists. Added objects are found by traversing the whole structure and adding all previously unknown objects. To know which objects are new the unique identifications of all visualised objects are stored and compared to each object encountered in the traversing process. If the unique identifier can't be found the object is new. The runtime system assigns a unique identification to every object, which is mainly used for serialising objects.

Solution

To see if the presented concept works and in order to gain some experience we have developed a prototype, which will be explained in more detail in this chapter. We decided to use Programmer's Open Workbench (Pow!) [Mühlbacher97] and the programming language Oberon-2 [Mössenböck92] for our tool. Pow! is a software development tool developed in one of our earlier projects.

Principally Pow! is designed for integrating various compilers for various programming languages. In our context however only the Oberon-2 compiler is relevant. This generates native stand alone programs for Windows 3.x,Win 95 and Windows NT. For the prototype we have used the 16 –Bit version simply because when this visualisation project was started the 32-Bit version was not yet fully tested. However this does not affect the underlying principles used here.

One of the major reasons to use Pow! is that we have implemented and hold the source code of the whole system including the compiler and linker. Particularly, having the source code of the compiler was important. As one can see later on we had to change the way the compiler generates type information. However it should be no problem to port the tool to any other environment and language if it provides all necessary information described in the following.

In contrast to programming languages like Java or C++ another terminology is used for Oberon-2. It doesn't have classes, it has special record types instead, which can be compared to classes. These record types have type bound procedures which are essentially what are called “methods” elsewhere. Record types can inherit fields and type bound procedures from other records which leads to the same class hierarchies as in other single inheritance programming languages. Also polymorphism is possible which is implemented by method tables [Mössenböck92], as in other programming languages. Therefore the programming language Oberon-2 should not be a restriction for proving the concept.

Heap Browser

The heap browser is responsible for gathering information about all objects dynamically created by parallel running programs. It provides the name of each object and the type of all records and their fields. To know which objects are currently allocated the heap browser scans the dynamic memory allocated by the runtime system. It asks the runtime system for the addresses of the allocated memory blocks and scans it knowing the exact structure of the heap.

After knowing all objects created so far the heap browser has to determine the type of each object. This is done by analysing the type information generated by the compiler. Any Oberon-2 compiler generates runtime data structures, which contain certain information about records. All this information is gathered in data structures called type descriptors. Every record has a pointer to its type descriptor, which is called the record's type tag. It is located just before the record's data and is invisible to the programmer. The type tag identifies the dynamic type of records.

The following example shows the record type CAR and a variable carP pointing to a record of type CAR. The statement NEW(carP) allocates a record of type CAR and carP points to its data as shown in figure 6. The type tag is located ahead of carP's data and points to the type descriptor of record type CAR.

```
TYPE
  CAR = RECORD
    Color: .....;
    Vendor: .....;
    ....
  END;
....
VAR carP: POINTER TO CAR;
....
BEGIN
  NEW(carP);
  ....
```

```
END;
```

Figure 6: Type tag and type descriptor

The Oberon-2 compiler of Pow! uses an extended structure. The type tag is placed at the same location and points to the type descriptor. The main difference is that the Oberon-2 compiler of Pow! provides precise information about each record field, especially the field's type and name. It is used for meta programming and garbage collection and is called runtime type information (RTTI). Equipped with that information the heap browser can determine the type and value of each field by analysing the RTTI.

After determining the type of each object and the name of each field the heap browser presents the found objects and allows the selection of objects. An example is shown in figure 7.

Figure 7: Object Selection Dialog

Visualisation Library

The visualisation library provides a set of standard classes, each of which are responsible for visualising exactly one type of object. Figure 8 shows some of them. A user may take them either unchanged or he or she can extend them for particular needs as explained earlier.

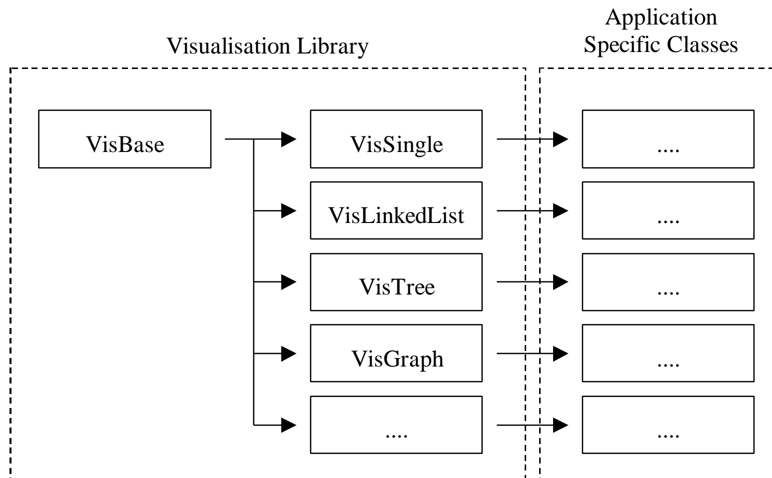

Figure 8: Visualisation Library

Details about the Visualisation System

The visualisation system is the last part of the whole work. It combines the other parts into a consistent tool and handles the communication between each part.

First of all it handles the whole user interface. Every time the user selects an object a new window is opened which shows the selected object. If the object is a root node of a data structure, all objects of the data structure are displayed in the same window. The pointers from one object to another are shown as arrows from one object to another.

Selecting an object for visualisation causes a call to the heap browser. More precisely the function SelectObject is called which returns the addresses of all selected objects in an open array. Then the visualisation system scans that array and determines the type name and module name of each address. With this information it can call the function GetDefault to get the default visualisation class for the specified object. Then the library which provides the default visualisation class should be loaded. Since the filename of the library is returned by GetDefault the visualisation has to call the API function LoadLibrary. After loading the library an instance of the default visualisation class can be created. The initialisation function of the visualisation object is called providing the address of the selected object.

```
VAR objects: ARRAY MAX_OBJECTS OF LONGINT;
Call SelectObject(..., objects);
FOR i := 0 TO LEN(objects) - 1 DO
  GetModuleName(objects[i], module);
  GetTypeName(objects[i], type);
  Strings.Copy(name, module);
  Strings.Append(name, ".");
  Strings.Append(name, type);
  IF GetDefault(name, class, library) THEN
    Windows.LoadLibrary(library);
    (* create object of class *)
    visObj.Init....(objects[i]);
  ELSE
    (* display error message *)
  END;
END;
```

The visualisation system holds a list of all visualisation objects. It distinguishes between simple objects and objects as instances of a more complex data structure, e.g. a polymorphic list. If a visualisation object represents a data structure the visualisation object itself is responsible for storing references to all visualisation objects of the data structure. The references between these visualisation objects are stored in an adjacency matrix M, which can represent every type of dynamic data structure from a simple linked list to a general graph. The elements M[i,j] are either empty or they contain the reference object. The latter is used to draw the reference (link, pointer) between the two objects i and j. Figure 9 shows the structure using a polymorphic list, consisting of objects a, b and c. The visualisation object A represents the list and for every object a corresponding visualisation object V(a), V(b) or V(c) is created. The adjacency matrix M maintains the references R from the object V(a) to V(b) and V(b) to V(c).

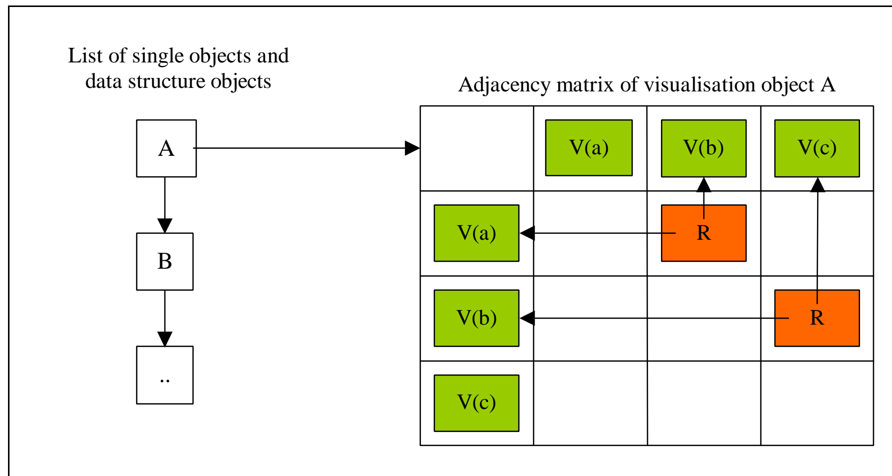

Figure 9: Adjacency matrix in visualisation objects

Figure 10 shows an example with a compiler written by another department. It compiles MiniModula programs [Rechenberg94] to a code for a special stack machine. It is used mainly for lectures to teach students implementing compilers. MiniModula is a simplification of the programming language Modula-2. The following simple program was used for the example:

```
MODULE Eratos;

VAR
  i,j,k : INTEGER;
  sieve : ARRAY (300) OF BOOLEAN;

PROCEDURE A;
BEGIN
END A;

PROCEDURE D;
BEGIN
END D;

BEGIN
 FOR i := 1 TO 10 DO
    FOR j := 1 TO 10 DO
      Write(i*j,5);
    END;
    WriteLn;
  END;
  A;
END Eratos.
```

Figure 10 shows the visualisation of the symbol list generated by the compiler for the above program during interpretation of the program. For the visualisation of the symbol list new visualisation classes were created. One shows the module information on top of the figure. The four small rectangles on the left side are drawn by a visualisation class showing the variables i, j, k and sieve. Another class is responsible for drawing procedures. The two rectangles at the right side are drawn by instances of that class.

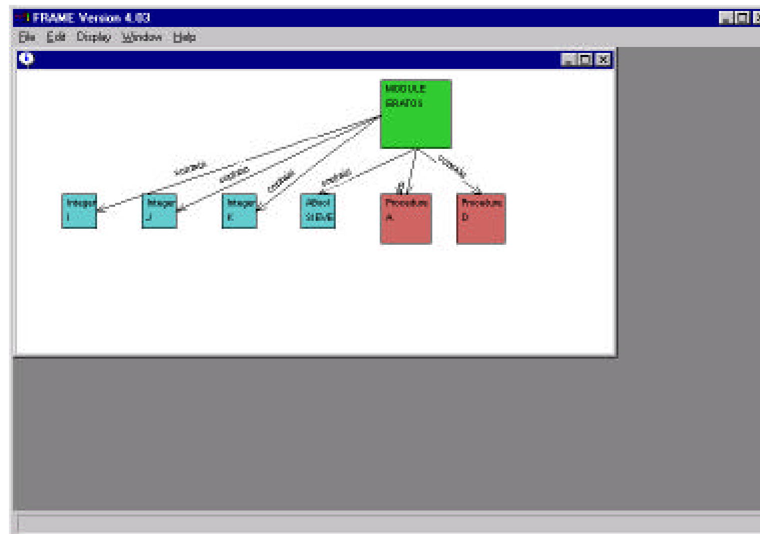

Figure 10: Visualisation of a Symbol List

Literature

[BaeckerMarcus86] Baecker, R. M. & Marcus, A. (1986). Design Principles for the Enhanced Presentation of Computer Program Source Text. In Proceedings of Human Factors in Computing Systems (CHI `88), (pp. 51-58). New York: ACM Press.

[BaeckerMarcus90] Baecker, R. M. & Marcus, A. (1990). Human Factors and Typography for More Readable Programs. Reading, MA: Addison-Wesley.

[Brown88] Brown, M. H. (1988b). Exploring Algorithms Using Balsa II. IEEE Computer, 21(5): 14-36.

[Brown91] Brown, M. H. (1991). Zeus: A System for Algorithm Animation and Multi-View Editing. In Proceedings of IEEE Workshop on Visual Languages, (pp. 4-9). New York: IEEE Computer Society Press.

[DingMateti90] Chen Ding, Prabhaker Mateti, A Framework for the Automated Drawing of Data Structure Diagrams, IEEE Transactions on Software Engineering, Vol. 16, No. 5, May 1990

[JerdingStasko94] Dean F. Jerding, John T. Stasko, "Using Visualisation to Foster Object-Oriented Program Understanding", Technical Report GIT-GVU-94-33, Georgia Institute of Technology, July 1994

[Kreuzeder99] Ulrich Kreuzeder, "Pow! - The programmers open workbench", Journal of Systems Architecture, Volume 45.11, May 1999

[Mössenböck92] Hanspeter Mössenböck, The Programming Language Oberon-2, Report 160, ETH Zürich, May 1992

[Mössenböck93] Hanspeter Mössenböck, Objektorientierte Programmierung in Oberon-2, 2. Auflage, Springer-Verlag, 1993

[Mühlbacher95] Jorg R. Mühlbacher, Bernhard Leisch, Ulrich Kreuzeder , "Programmieren mit Oberon-2 unter Windows", Hanser, 1995

[Mühlbacher97] Jörg R. Mühlbacher, Bernhard Leisch, Brian Kirk, Ulrich Kreuzeder, "Oberon-2 Programming in Windows", Springer, 1997

[Rechenberg94] P. Rechenberg, H. Mössenböck, Ch. Reichenberger, "MiniModula2", Abteilung für Software, Institut für Informatik, 10. Auflage, 1994

[ReissMeyers93] Steven P. Reiss, Scott Meyers, FIELD Support for C++, Report CS-93-03, Department of Computer Science, Brown University, February 1993

[Stasko90] Stasko, John T., "TANGO: A Framework and System for Algorithm Animation", IEEE Computer, Vol. 23, No. 9, September 1990, pp. 27-39.