

Enriching Program Comprehension for Software Reuse

Elizabeth Burd, Malcolm Munro

The Centre for Software Maintenance
University of Durham
South Road
Durham, DH1 3LE, UK

Abstract

This paper describes the process of code scavenging for reuse. In particular, we consider enriching program comprehension for the identification and integration of reuse components by information abstraction and the use of graphical representations. The requirements of good reuse candidates are described, and then a description of a process of identifying and preparing for their reengineering into reuse units is given. In particular, we describe two main activities: the identification of units; and the definition of the units user interface. Initially, the identification of reusable units applies some of the methods from RE² but is extended with the use of graph simplification procedures. The identification process is based on the calling structure of the code. Secondly, data analysis is performed on the identified reuse candidates. The data analysis process provides an indication of the potential use of the component and the effort required to make the candidate reusable.

This paper provides examples and results from a number of case studies which have been used to evaluate this work. Our work relies heavily on there being communication between technical and non-technical staff. We achieve this through the use of graphical representation and thus results are displayed graphically where applicable.

1. Introduction

Reuse has long been thought of as a way of improving productivity and maintaining, or even improving, quality standards [Biggerstaff89, Freeman87, Tracz88]. In addition, it is believed that reuse based software development can reduce the costs of maintenance because the nature of such software is more modular and therefore software updates are more localised. However, there is insufficient specifically designed reusable

software for many domains, so developers must rely on updating existing software if they are to benefit from reuse.

The process of finding and using software which has not been designed for reuse is often termed software scavenging. When scavenging is adopted, frequently, modifications (therefore maintenance) have to be performed on the software to make it reusable. However, if such maintenance issues can be overcome, the advantage of such an approach is the availability of large quantities of potentially reusable software and thus large overall potential cost saving.

For software to be reusable it must exhibit a number of features that directly encourage its use in similar situations. The following finding by Wood [Wood89] list the important qualities of reusable software. Reusable software should offer:

- Environmental independence - components should be reusable irrespective of the environment in which they were originally created
- High cohesion - components should implement a single operation or a set of related operations
- Loose coupling - components should have minimal links to other components
- Adaptability - components should be able to be customised so that they will work in a range of similar situations
- Understandability - components should be easily understandable so that users can interpret their functionality

It is believed that ideally the developer should be able to find large fragments of source code quickly that can then be reused without significant modification. However, all code proposed for reuse needs first to be assessed to evaluate the necessity for modifications to make it reusable. Furthermore, an evaluation needs to be applied to investigate whether the software reuse candidates comply with the Wood's quality issues.

In practice, the overall effectiveness of the scavenging process is severely restricted by its informality for the following reasons:

- a programmer can only reuse code fragment he has knowledge of
- there is no systematic way to share fragments
- identification and integration are inefficient and they require thorough understanding of the code

The first two issues are concerned with the setting up of a reuse environment where a repository of information available to support the reuse process. Thus with such support information is available for the systematic sharing of components there is an increased knowledge of the existence of reuse code. Work in this area has included reuse library classification mechanisms [Prieto-Diaz85] and the identification and retrieval of appropriate software components [Albrechtsen92].

This paper will discuss the issues raised in the third reason for the lack of reuse successes; the understanding of code for the identification and integration of reusable components. This paper describes the results of work we have recently completed involving assessing the feasibility of obtaining reusable components from legacy COBOL code. While many of our techniques are applicable to most software development languages, we investigate the usability of COBOL since it represents the largest set of legacy systems. The scenario of this work results from a realisation that many business applications perform the same operations repeatedly throughout a number of applications. Therefore, if this repeated functionality can be extracted from the source code and reengineered into a new reuse module the cost of future developments can be reduced and benefits will be gained from reduced maintenance. This paper describes our analysis process using actual business applications. The examples we will give are from a set of 12 case studies ranging from 3,000 to 40,000 lines of code excluding copy code. Each of the programs are from business applications and are still in use.

Throughout this paper we describe the process by which reuse candidates are identified and how their suitability as reuse candidates is evaluated. We describe analysis of the calling structure of the code and the identification of data interactions as the basis of designing an interface for the reuse modules. At each stage of the analysis process we describe how the process enhances understanding to assist the reengineering process of the interfaces. This understanding process is supported with the use of graphical representations. This aids the communication process between the users and maintainers of the code under analysis. Examples of the graphical representations that we use are provided throughout this paper.

The paper considers three aspects of the reuse process. Section 2 describes the identification of potential reuse candidates from existing legacy code. The integration of the components, and, in particular, our approaches to investigate the simplification of reuse candidate interfaces, is described in section 3. Finally, section 4 concludes with the results of our work and suggests areas of further work which will offer improvements to our existing approach.

2. Identifying Reusable Components

In section 1 we indicated the lack of reusable components designed specifically for reuse. For this reason it is necessary to find components from existing software applications. In order to locate such components we use a number of approaches from the RE² paradigm [Canfora95, Cimitile95, De Lucia95, Tortorella94]. The RE² paradigm provides many techniques for reengineering existing systems. Specifically we use RE² to investigate the calling structure of the code (COBOL PERFORMs) to generate a 'PERFORM graph', and the dominance relation of the calling structure between procedures (COBOL SECTIONs) to generate a 'dominance tree'. Using the largest and most complex of the examples we have analysed, we will describe the results of this analysis for the identification of reusable components. The benefits of each technique towards aiding understanding of the reuse candidate are also now explored.

A PERFORM graph is an example of a call-directed-graph. A PERFORM graph is constructed by analysing the PERFORMs for each SECTION of the COBOL program. Within each of the sample programs it was the convention to structure the code with SECTIONs. Each SECTION contained two PARAGRAPHS, one acting as an EXIT function. This convention ensures that there is no 'fall through' between SECTIONs. For each PERFORM in a SECTION a link is made between the calling and the called SECTION. In COBOL such an assessment of the calling structure can be complicated with the use of statements such as GOTOs and through implicit fall through. However, we were fortunate that the code used within the case study was well structured and did not support such features. The results of the PERFORM analysis can be seen in figure 1.

Figure 1 gives an indication of the complexity of the software being analysed. Within the 40,000 lines of code there are 230 SECTIONs and between the SECTIONs 553 different relationships 932 in total (when including the number of times SECTIONs call the same SECTION is taken into account). A number of graph simplification procedures can be used to

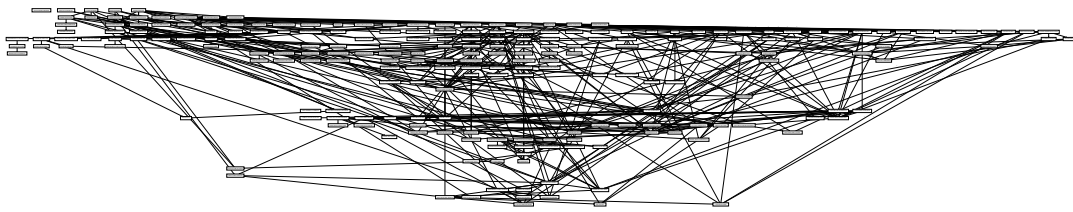


Figure 1: PERFORM graph

improve the understandability of the PERFORM graph including, for example, removing SECTIONS which are only concerned with error checking (frequently such SECTIONS mask the true functionality of the program).

The results of removing 3 types of nodes can be seen in table 1. The number of corresponding links which are removed along with the removal of the different types of nodes is also indicated.

Type of SECTION (node) Removed	Number of nodes	Number of links
Error handling	5	80
Database checks	2	26
Message composition	2	10
Total	9	116

Table 1: The results of the graph simplification process

The simplification process in this case has resulted in the removal of very few nodes only 4% of the total number of SECTIONS within the code. However, this has resulted in a 21% reduction in the number of links between the remaining SECTIONS.

In summary, the PERFORM graph provides an indication of the complexity of the code being analysed. In addition, 'problem SECTIONS', those with a high degree of connectivity to other SECTIONS, can be pinpointed for investigation as, frequently, those nodes represent the SECTIONS providing error checking and mask the system's functionality.

From the PERFORM graph we are able to generate a dominance tree. The PERFORM graph is initially turned into a call-directed-acyclic-graph (CDAG). The graph is obtained by collapsing every strongly connected subgraph into one node. Thus all nodes involved in a recursive cycle are grouped to form a single node of the graph. From this the dominance tree is constructed using a method devised by Hetch[Hetch77]. According to Hetch in a CDAG a node px dominates a node py if and only if every path from the initial node x of the graph to py spans px .

In order to find potential reusable candidates a tree of strong direct dominance between nodes is considered. In a CDAG there is a relation of strong direct dominance

between the nodes px and py if and only if px directly dominates, and is the only node that calls py .

The results of this analysis is shown in figure 2. Those nodes which are not strongly dominant are darkly shaded within the figure.

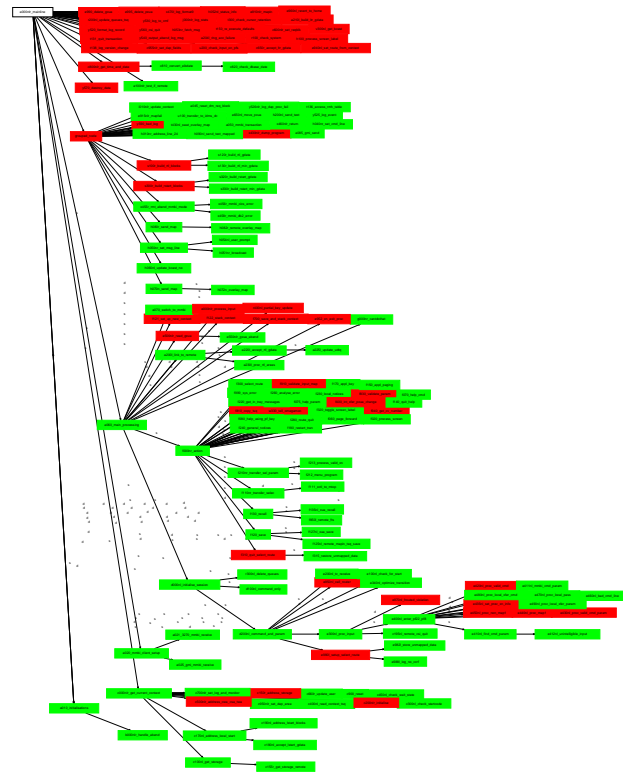


Figure 2: The dominance tree

The subgraphs within the tree which do not include the darkly shaded nodes represent the potential reuse candidates. From the code analysed above, 11 potential reuse candidates can be generated. The size of these candidates ranges from a few hundred lines to several thousand lines of code per reuse candidate. However, in other examples we have found up to 20 from a code module of around this size. Our findings have left us to conclude that this is not only the size of the initial code module which indicates the number and size of reuse candidates that are obtained from this analysis, but the complexity of the interaction between SECTIONS. For this reason, moving 'non-functional' SECTIONS (i.e.

error checking) before generating the dominance tree does help to identify larger components. However, the implication of removing the SECTIONS for a purpose other than simply aiding the understanding process needs to be considered carefully.

The PERFORM graph and dominance tree provide a communication mechanism through their graphical representation. From these discussions we can begin to establish which candidates should be considered further. Therefore, we consider which function or sets of functionality are provided by the identified candidates which is to be most useful for later reuse. This process is concept assignment. Further details on how to perform this process can be found in Biggerstaff [Biggerstaff94]. Predictions of potential future reuse are dependent upon the future business strategy, but factors like the effort required to reengineer the component into a reusable form need also to be considered. Therefore, the graphical representations we use are an important aid to assist discussion between management or systems procurement and technical staff.

The graphs provide a fairly high level view of potential components. This enables the rejection of candidates which are unsuitable i.e. those whose functionality is deemed unsuitable. However, the work we have described so far does not provide any details as to the reengineering effort required to make the candidates reusable. For this reason, it is important to consider data analysis. However, the cognitive load for such a process is large. Therefore, we deliberately delay this detailed analysis until only the most promising components, selected from the use of the dominance tree, remain.

3. Integrating Reusable Components

The ease of integration of components relates to the qualities of high cohesion and loose coupling. It is desirable to achieve high cohesion within the reuse module so the module only implements one, or a group of similar features. This point was considered in the previous section. However, tied with high cohesion, is loose coupling. Loose coupling represents the interaction between the reuse candidates and the application under development. To investigate the interactions we consider the data dependencies for each of the reusable components. The data dependencies between reuse components form the component's user interface, so whenever possible, we seek to reduce unnecessary interactions and therefore simplify its user interface.

We perform the analysis of the data items at two levels. Initially, data analysis is performed within the reuse candidate itself, and, secondly, the analysis is performed to investigate the boundary between the perspective

reuse candidate and the remaining code that the candidate will be separated from.

The initial analysis within the reuse candidate performs the following activities:

1. C. R. U. D. - data items are categorised using SSADM (Structured Systems Analysis and Design Method) data groupings i.e. as to whether they are Created, Read, Used and/or Deleted within the reuse candidate.
2. Data inter-relationships between typing - where subtyping is used but reference is made within data items of the same type to both super and sub-types.
3. Data inter-relationships by value sharing - where enumerated types are defined and value overlapping is recorded.
4. The use of the REDEFINES relationship - where one set of types or data items is used to redefine another set of types or data items.

The secondary analysis between the reuse candidate and the remainder of the code involves the following activities:

1. C. R. U. D. - data items which are Created, Read, Updated, or Deleted within the reuse candidate but used within the remainder of the code, or vice versa, are recorded for further analysis.
2. Data inter-relationships between typing - a record is made where supertypes are used within the reuse candidate but its sub-types are used within the remainder of the code, or vice versa.
3. Data inter-relationships by value sharing - a record is made where data items involving value overlapping are used within the reuse candidate and also within the remainder of the code.
4. The use of the REDEFINES relationship - where REDEFINES statements forces relationships between the reuse candidate and the remainder of the code.

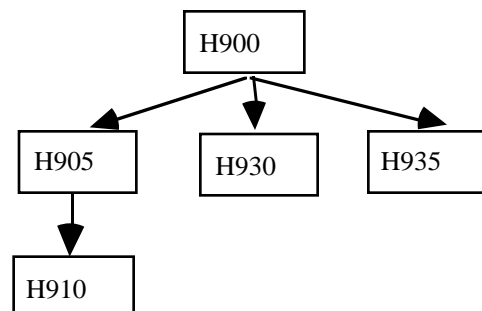


Figure 3: A strong dominance subgraph reuse candidate

```

01 INPUT-DATA.
05 INPUT-DATE PIC 9(8)
05 INPUT-RATE-R1 REDEFINES INPUT-DATE
10 FILLER PIC 9(2)
10 INPUT-YEAR PIC 9(2)
10 INPUT-MONTH PIC 9(2)
10 INPUT-DAY PIC 9(2)

```

Table 2: Sample COBOL code

To demonstrate how we make use of the above activities and how these are used to reduce the interactions of the reuse candidates, therefore assisting the integration process, we give examples from one of the medium sized software modules analysed. We select a reuse candidate of around 1,000 lines of code which is composed of 5 SECTIONS. The dominance tree for the subgraph to be analysed is shown in figure 3.

All data items of the reuse candidate are then analysed to group them according to the categories proposed in activity 1, above. The table (table 3) below shows the results of this analysis process. Within this paper we are particularly concerned with the data items which are updated and those which are only read. This information provides an indication of how candidate reuse modules interact with each other. In table 3 we have not included the IDMS database statements as for the purpose of this paper we restrict our analysis to COBOL source code. We therefore see little use, as one might expect, of creation and deletion of data items. However, we have included the results here for completeness.

	H900	H905	H910	H930	H935	Total
Created	1	0	0	0	0	1
Read	56	55	10	10	2	133
Update	14	20	6	6	1	47
Deleted	0	0	0	0	0	0

Table 3: H900 subgraph data usage

Unfortunately, quite a high proportion of the data items of the reuse candidate seem also to be used throughout the remainder of the code. Thus, the reuse candidate without reengineering would have a complex interface. We will investigate ways of reducing the complexity of the interface throughout the remainder of this section.

By categorising data items using SSADM's Creation, Read, Updated and Deleted, unnecessary interactions between reuse units data accesses can be identified. For instance, those SECTIONS where data items are defined but not used should be identified and such definitions moved to the SECTION where the data is used. The long term benefit of making such changes can be identified by assessing the reduction in the number of data interactions between SECTIONS and thus a decrease in the complexity of the reuse candidates interface. This

helps to gain an indication of the costs and benefits gained for the approach. In the reuse candidate, one candidate data item for relocation was identified.

Activity 2 identifies the need to address relationships between data items through typing. For instance, in COBOL we find the construct which is shown in the code sample of table 2.

From this we can derive the following *consists-of* data relationships shown in figure 4.

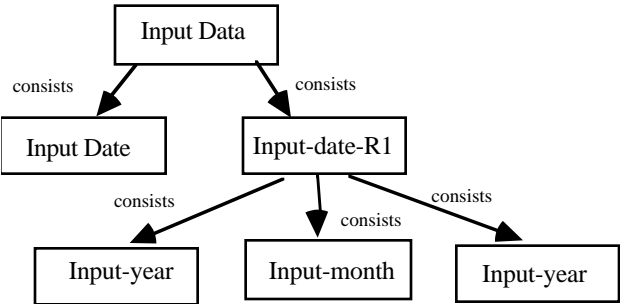


Figure 4: An example of the consists relationship from COBOL data typing

For figure 4 we can identify that Input-data consists-of Input-date and Input-date-R1. In some cases not all of the type hierarchy will be present within the reuse candidate code. The absence or presence of part of the type hierarchy from the reuse candidate can be represented graphically by shading. It is important to examine how the typing is used within the candidate to examine fully the relationships between it and the remainder of the code. The results of this analysis process can be seen in figure 5. The shaded boxes represent those data items which are referenced within the reuse candidate.

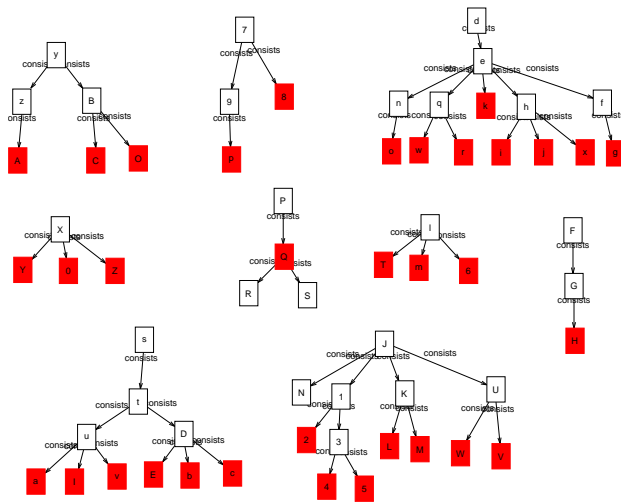


Figure 5: The consists relationship for the reuse candidate

From figure 5, it can be seen that there are no internal inter-relationships between the data items of the reuse candidate. Inter-relationships would be identified by the existence of two shaded boxes within one path of the consists trees. We need therefore only to consider the relationships of the data items between the reuse candidate and the remainder of the code.

When analysing the interaction of data items between the reuse candidate and the remainder of the code, we need to consider two such relationships. We need to record how the remainder of the code effects the reuse candidate and also record how the reuse candidate effects the remainder of the code. From the analysis of figure 5 we can conclude that the reuse candidate is potentially effected by write operations on sub-types whose supertype is read by the candidate and also by supertypes whose subtypes are updated by the remainder of the code. From the analysis of the code, we have identified only two interacting data items which must be recorded. These are the grandchildren on *P*, nodes marked *R* and *S*. No write operations are performed on the supertypes. However, all write operations within the reuse candidate will effect the remainder of the code.

```

01 DATA-INPUT.
  05 -DAY-OF-WEEK PIC 9.
    88 MONDAY PIC 9(1) VALUE 1.
    88 TUESDAY PIC 9(1) VALUE 2.
    ....
    88 SATURDAY PIC 9(1) VALUE 6.
    88 SUNDAY PIC 9(1) VALUE 7.
    88 WEEKEND PIC 9(1) VALUES ARE 6 7.
  
```

Table 4: Code sample with Enumerated typing

Overlapping values (activity 3) are evident where use is made of enumerated typing. For instance, in the code sample shown within table4.

In the above example the last three data types overlap. We are able to represent these overlapping values graphically as an extension of the consists graph (figure 6). In this example the weekend data type groups its two overlapping values. The notation can be further extended when the reuse candidate does not comprise all the possible values, by adding box colour coding.

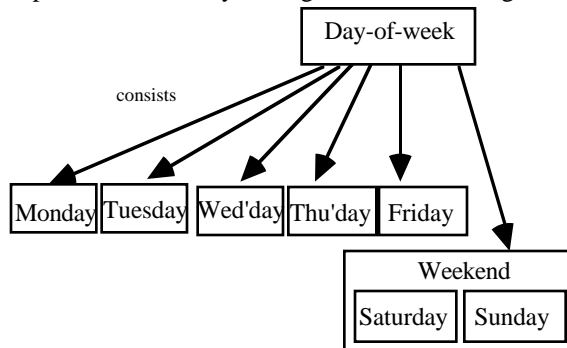


Figure 6: An example of overlapping values within the consists graph

In our case study, 4 sets of overlapping values have been identified, but only 1 set is involved within the subgraph. The result of the data analysis has found that there is use made of some of the data values within the reuse candidate consists graph as well as the remainder of the code. During the above example, we have found the shaded data items (figure 7) to be used outside the reuse candidate.

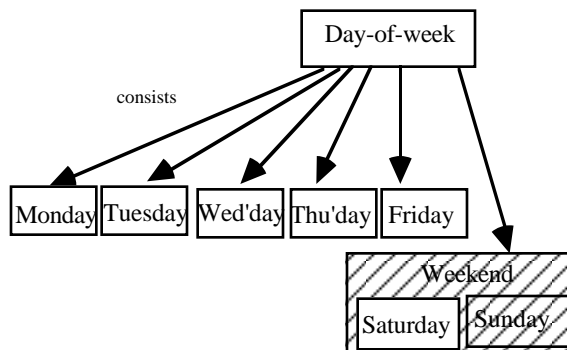


Figure 7: Values used within the remainder of the code

Therefore this procedure provides the justification of why we must record both the Weekend and Sunday and also Day-of-week data items, as each are potentially affected by the remainder of the code. This indicates a further area where the simplification of the user interface could be achieved by reengineering.

Further data interrelations also exist through the redefines relationship (activity 4). An example of the

usage of such a construct can be found in the typing sample (activity 2). i.e.

05 INPUT-RATE-R1 REDEFINES INPUT-DATE

The effect this has on the consists graph (figure 4) is shown in figure 8. In addition, in some cases it is important to consider the reverse relationship 'is-redefined-by'. In the example below, we can identify this relationship being Input-date *is-redefined-by* input-date-R1. We will now consider the results of the analysis for these two relationships.

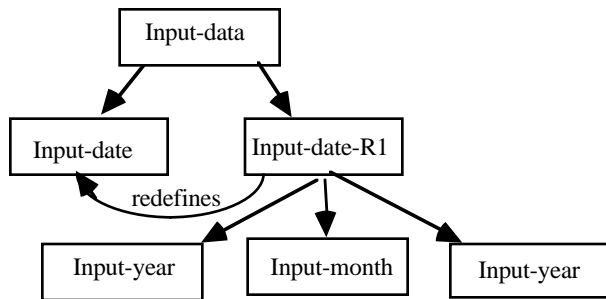


Figure 8: An example of the redefines relationship

Within the reuse code from which we identified the reuse candidate, we have identified 18 REDEFINES relationships. Within the reuse candidate, no data items are redefined internally or outside the subgraph. Furthermore, the same result was found with the is-redefined-by relation. That is, no data items within the remainder of the code are redefined-by data items within the subgraph.

Throughout this section we have described how detail analysis is performed to identify the data interactions between the reuse candidate and the remainder of the code. Each of the four activities enhance the understanding process towards indicating the cohesion and coupling within the candidate. The activities gradually provide more and more information about the reuse candidates so that initially their usefulness and finally the effort to reengineer the candidates can be assessed. An effort is made to pinpoint areas where reengineering should concentrate to reduce the complexity of the components interface and so enhance its reusability. Furthermore, this is an important first step towards making the reuse candidate environmentally independent and adaptable for reuse in many different circumstances. Its reusability is also dependent upon the re-user understanding the functionality of the component. Graphical representations of the calling structure and data interactions are used whenever possible to allow visual identification of the unit's functionality.

4. Simplifying the Graph Structure

This paper has described a process by which reuse candidates can be identified and evaluated. The identification process concentrates on the structure of the code from a high level, i.e. the calling structure. This process identifies a set of potential candidates which can then be evaluated by reuse support staff to select which sets of functionality will be most reusable for future applications development. Our case studies have shown that using software managers to perform tasks of concept assignment is a good starting point to evaluating the candidates suitability for reuse. We have found that our graphical representations very much support this process as a way of providing an abstraction of the source code. In most cases the managers were able to gain sufficient information from the SECTION names to be able to provide a brief description of the functionality of the candidate without having to refer to the source code. These descriptions of the candidates functionality could then initiate the discussion concerning the potential reusability of each of the candidates.

The selection of appropriate reuse candidates goes beyond simply investigating the functionality provided. If a significant degree of reengineering work must be performed on the candidate to make it reusable then it is important to assess the cost of performing such work along with the potential gains for having the reuse module available for future application developments. The most significant cost of the scavenging process is the encapsulation of the reuse candidate to define a simple user interface to ensure that it is usable in new applications. The information for such decisions is provided through our data analysis which we have described in section 3. The details provided by performing the described activities serve to enhance the knowledge gained from the SECTION analysis and discussions with management to assign function descriptions to the candidates. Each of the four activities provides important details of how the data from the reuse candidate interacts with its environment. The data interactions between the reuse candidate and the original code can be overcome by defining the data items within the user area. However, if possible data interactions should be avoided as this helps to keep the user interface free from unnecessary complexity.

From performing the SECTION and data analysis it is possible to gain a full and detailed understanding of the extent of the work involved in providing an interface for the reuse candidate. This can then be used to support the process whereby reuse candidates can be evaluated for their potential reusability. Thus, throughout this paper we have described the process whereby candidates can be selected and assessed.

Two main factors of Wood's [Wood89] reuse qualities remain to be considered: those of adaptability and environmental independence. These issues relate to the adaptation of reuse candidates to make them potentially reusable in more applications by improving their generality. The details gained from performing these activities described within this paper provide the backbone of the information required to perform the generalisation tasks. However, to investigate such issues, this work needs to be coupled with reuse case scenarios. This is an area of work which is currently ongoing.

At present this method is supported by pre-prototype tools. The result of using these tools has been to demonstrate that tool support can serve to automate many aspects of the method, providing it is also supported by human understanding, for instance, the concept assignment process. The result has been that the lengthy tasks can be completed automatically, such as the SECTION analysis and providing abstractions of the reuse candidate, to aid the human understanding process. While this process has worked well, commercial strength tools must be developed to support the analysis of very large applications.

Acknowledgements

The authors wish to acknowledge, and thank, British Telecommunications for their assistance and financial support for this project.

References

- [1] Albrechtsen H., 'Software Information Systems: information retrieval techniques', in *Software Reuse and reverse Engineering Practice*, P.A.V. Hall (ed.), Chapter 6, UNICOM Applied Information Technology 12, Chapman & Hall, 1992
- [2] Biggerstaff T.J., Perlis A.J., (eds), 'Software Reusability', Vol 1 and 2, ACM Press, Addison Wesley, 1989
- [3] Biggerstaff T.J., Mitbender B.G., Webster D.E., 'Program Understanding and the Concept Assignment Problem', *Communications of the ACM*, May 1994
- [4]. Canfora G., Cimitile A., Visaggio G., 'Assessing Modularization and Code Scavenging Techniques', *Journal of Software Maintenance*, Vol 7, No 5, October 1995
- [5] Cimitile A., Visaggio G., 'Software Salvaging and the Call Dominance Tree', *Journal of Systems and Software*, vol 28, No 2, February 1995
- [6] De Lucia A., 'Identifying reusable Functions in Code Using Specification Driven techniques', M.Sc Thesis, University of Durham, 1995
- [7] Freeman P., (ed), 'Tutorial on Software Reusability', IEEE Computer Society Press, New York, 1987
- [8] Hetch M.S., 'Flow Analysis of Computer Programs', Elsevier, North Holland, 1977
- [9] Prieto-Diaz R., 'A Classification Scheme', PhD Thesis, Department of Information and Computer Science, University of California at Irvine, 1985
- [10] Tortorella M, E., 'Identification of Abstract Data Types in Code', MSc. Thesis, University of Durham, 1994
- [11] Tracz W., 'Tutorial on Software Reuse: Emerging Technologies' IEEE Computer Society Press, New York, 1988
- [12] Wood M, 'Software Function Frames: an approach to the classification of reusable software through conceptual dependency', PhD Thesis, University of Strathclyde, January 1989