A method for re-modularising legacy code

Elizabeth L. Burd

A Method for Re-modularising Legacy Code

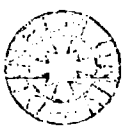
Elizabeth L. Burd

Ph.D. Thesis

The Research Institute in Software Evolution

Department of Computer Science

University of Durham



June 1999

Abstract

This thesis proposes a method for the re-modularisation of legacy COBOL. Legacy code often performs a number of functions that if split, would improve software maintainability. For instance, program comprehension would benefit from a reduction in the size of the code modules. The method aims to identify potential reuse candidates from the functions re-modularised, and to ensure clear interfaces are present between the new modules. Furthermore, functionality is often replicated across applications and so the re-modularisation process can also seek to reduce commonality and hence the overall amount of a company's code requiring maintenance.

A 10 step method is devised which assembles a number of new and existing techniques into an approach suitable for use by staff not having significant reengineering experience. Three main approaches are used throughout the method; that is the analysis of the PERFORM structure, the analysis of the data, and the use of graphical representations. Both top-down and bottom-up strategies to program comprehension are incorporated within the method as are automatable, and user controlled processes to reuse candidate selection.

Three industrial case studies are used to demonstrate and evaluate the method. The case studies range in size to gain an indication of the scalability of the method. The case studies are used to evaluate the method on a step by step basis; both strong points and deficiencies are identified, as well as potential solutions to the deficiencies.

A review is also presented to assesses the three main approaches of the methods; the analysis of the PERFORM and data structures, and the use of graphical representations. The review uses the process of software evolution for its evaluation using successive versions of COBOL software. The method is retrospectively applied to the earliest version and the known changes identified from the following versions are used to evaluate the re-modularisations. Within the evaluation chapters a new link within the dominance tree is proposed as is an approach for dealing with multiple dominance trees. The results show that each approach provides an important contribution to the method as well as giving a useful insight (in the form of graphical representations) of the process of software evolution.

Acknowledgements

This work has been carried out at the University of Durham, Department of Computer Science, Research Institute in Software Evolution (RISE). The work described was conducted within the BT funded IDENT and Futures projects and the EPSRC funded Release project.

My thanks go to Dr Vernon Armitage and Professor Michael Prestwich for their help with my registration. To my fellow colleagues of RISE for their helpful comments and encouragement and to my forever patient supervisor Professor Malcolm Munro. Last but not least to Sarah, Claire and Debs for without their support I would not have got this far.

Copyright

Declaration

The material presented on this thesis is the product of my own independent research carried out at the University of Durham under the supervision of Professor Malcolm Munro. It was undertaken within the BT funded IDENT and *Futures* project and the EPSRC funded *Release* Project. The work described in this thesis has also been reported, in part, in the following:

- Aylett M., Burd E.L., Munro M., 'Identification and Encapsulation of Reusable Functions', *Durham Technical Report*, 1/96, 1996.
- Burd E.L., Munro M., Wezeman C., 'Analysing Large COBOL Programs: the extraction of reusable modules', published in *Proceedings of the International Conference on Software Maintenance*, California, IEEE Press, 1996.
- Burd E.L., Munro M., 'Identification of Reusable Modules', published in the proceedings of the *2nd UK Program Comprehension Workshop*, Durham, 1996
- Burd E.L., Munro M., Wezeman C., 'Extracting Reusable Modules from Legacy Code: Considering issues of module granularity', published in *Proceedings of the 3rd Working Conference on Reverse Engineering,* California, IEEE Press, 1996.
- Burd E.L., Munro M., 'Enriching Program Comprehension for Software Reuse', published in the *Proceedings of the International Workshop on Program Comprehension: IWPC'97*, IEEE Press 1997.
- Burd E.L., Munro M., 'Investigating the Maintenance Implications of the Replication of Code', to be published in the *Proceeedings of the International Conference on Software Maintenance; ICSM'97*, IEEE Press 1997.
- Burd E.L., Munro M., The Implication of Non-functional Requirements for the Reengineering of Legacy Code', published in *Proceedings of the 4th Working Conference on Reverse Engineering,* Amsterdam, IEEE Press, 1997.

- Burd E.L., Munro M., 'Examining Software Evolution to Support Reengineering of Legacy Code', published in the *Proceedings of the International Workshop on the Principle of Software Evolution,* 1998
- Burd E.L., Munro M., 'A Method for the Identification of Reusable Units Through the Reengineering of Legacy Code', published in the *Journal of Software and Systems*, December, 1998
- Burd E.L., Munro M., 'Reengineering Support for Software Evolution: an evaluation through case study', published in the *Proceedings of the Computer Software and Application Conference, COMPSAC'98*, IEEE Press, 1998
- Burd E.L., Munro M., 'Investigating Component Based Maintenance and the Effect of Software Evolution: a reengineering approach using data clustering', published in the *International Conference on Software Maintenance:* ICSM'98, IEEE Press, 1998
- Burd E.L., Munro M., 'Assisting Human Understanding to Aid the Targeting of Necessary Reengineering Work', published in the proceedings of *the International Workshop on Reverse Engineering*, IEEE Press, 1998
- Burd E.L., 'Reuse Reengineering', published in the proceedings of *the Ingeniería de Software y Reutilización: Aspectos Dinámicos y Generación Automática*, ISBN 84-8408-0237-7
- Burd E.L., Munro M., 'Transforming Legacy Code', published in the proceedings of the *International Workshop on Transformation Systems: STS'99*, ICSE'99 Workshop, 1999
- Burd E.L., Munro M., 'Characterising the Process of Software Change' published in the proceedings of the *Workshop on Principles of Software Change and Evolution: SCE'99*, ICSE'99 Workshop, 1999
- Burd E.L., Munro M., 'Evolution as an Instrument of Software Change', *Durham Technical Report*, tr_02-99, 1999
- Burd E.L., Munro M., 'Using Evolution to Evaluate Reverse Engineering Technologies', published in the proceedings of *Empirical Studies on Software Development and Evolution: ESSDE'99*, ICSE'99 Workshop, 1999
- Burd E.L., Munro M., 'Visualising Software Evolution to Support Reengineering of Legacy Code', to be published in *Software Engineering Notes*, ACM Press, 1999
- Burd E.L., Munro M., 'Object Identification', *Durham Technical Report*, tr_04-99, 1999
- Burd E.L., Munro M., 'Evaluating the Use of Dominance Trees for C and COBOL', to be published in the *Proceedings of the International Conference on Software Maintenance; ICSM'99*, IEEE Press, 1999
- Burd E.L., Munro M., 'Evaluating the Evolution of a C Application', to be published within the Proceedings of the *International Workshop on the Process of Software Evolution*, 1999

Contents

Figures

Tables

Chapter 1. Introduction

Blum [Blum96] states in his book regarding the future issues of software development that, in the future, reuse will have a different character. He states that:

> *There already is enormous investment in software, and there is little likelihood that the software can ever be replaced. It would be too costly to reprogram, and few understand exactly what the current systems do. Thus, despite their imperfections, we must learn to reuse key portions of the available (legacy) systems.*

Thus, Blum's predictions define the objective of this work: to define an approach for the identification, extraction and encapsulation of potential reuse candidates from legacy systems.

The availability of such candidates reduces the overall costs for initiating a **reuse** program by acquiring candidates that are already developed. Furthermore, this process also assists **maintenance** in allowing selective improvements to be made to the interface of frequently used functionality. Thus, the approach adopted is one of **reuse reengineering**.

The basic reuse reengineering process is as follows:

1. Large 'multi-functional' legacy systems are identified for their provision of functions used frequently within a company's applications.
2. These legacy systems are broken down into smaller units with a single functionality.
3. Well defined and, preferably, simplified interfaces are formed from the original complex interactions.
4. The new units are used to replace the legacy systems, thus assisting future maintenance.
5. The new units are also available to use within developing systems. Thus they are available for reuse.

This process is represented graphically in figure 1.1.

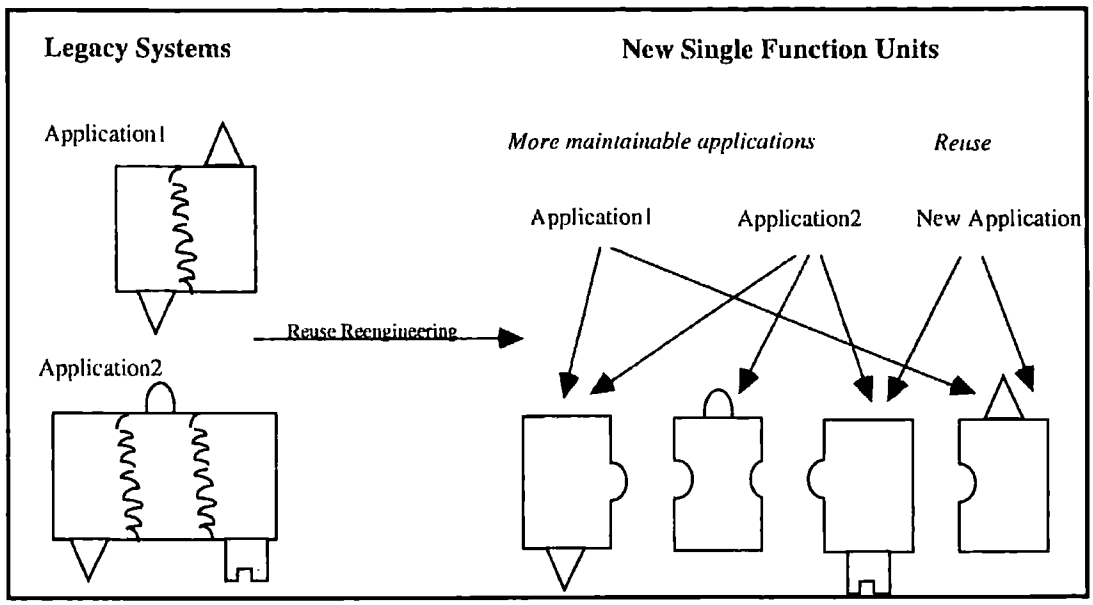


Figure 1.1 The reuse reengineering process

In Figure 1.1 Application1 is split into two reuse candidates. One of the reuse candidates has the same functionality as a reuse candidate identified within Application2. This reuse candidate is then reused within new versions of Application1 and Application2 and a new application.

1.1. The Software Maintenance Problem

Software maintenance has been defined within the ANSI standard [ANSI83] as:

> *"the modification of software products after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed environment."*

Maintenance is a costly activity. Parikh and Zvegintzov [Parikh83] indicate that software maintenance consumes 50% of all computer resources and research by Boehm [Boehm75] has shown that maintenance costs can be up to ten times those of an initial development.

The definition of software maintenance implies that there are a number of reasons why the maintenance process should be initiated. These differing reasons result in four categories of maintenance activities. These are:

- Perfective maintenance - this involves improving functionality of software in response to a user's defined changes.
- Corrective maintenance - this process involves the correction of errors that have been identified within the software.

- Adaptive maintenance - this process involves the alteration of the software which is due to changes within the software environment.
- Preventative maintenance - this involves updating the software in order to improve upon its future maintainability without changing its current functionality.

The costs of the maintenance processes are not distributed evenly across all categories. Studies by Leintz and Swanson [Lientz80] show that 50% of the total maintenance costs can be attributed to perfective maintenance, 25% for adaptive maintenance, whereas only 21% of the total costs are attributed to corrective maintenance and 4% for preventive maintenance.

Despite the cost implications, software maintenance it is generally perceived as having a low profile within the software community. Management teams have often in the past placed little emphasis on maintenance related activities. Small advances have been made in combating these problems, and high profile maintenance projects such as the year 2000 problem have been successful at highlighting the issues.

Software maintenance is made difficult by the age of the software requiring maintenance. The age of the software means that documentation has often been lost or is out of date. Furthermore, issues of staff turnover and constant demands for changes due to user enhancements or environmental changes exacerbate the problems. In addition, constant perfective and corrective maintenance, which is not supported by preventative maintenance, has a tendency to make the software more difficult to maintain in the future.

Software maintenance is difficult to perform on highly maintained software due to the ripple effects on the other parts of the system. Ripple effects are the changes that become necessary due to maintenance on another part of the code. Ripple effects can occur at the design and specification levels and may additionally involve changes in the documentation. The task of assessing these ripple effects is called impact analysis.

The availability of well defined interfaces between separate areas of functionality within the code can go a long way towards reducing ripple effects and therefore reducing the overall costs of changes. Thus, the reuse reengineering process is in fact a form of preventative maintenance.

1.2. Research Problems

This thesis concerns the generation of reuse candidates from legacy systems. In particular, it considers the identification of the candidates from any available information (such as manuals) but specifically relies upon the source code. A number of existing projects have contributed to this work. However, no existing approach has managed to completely solve the problems associated with the identification

process. This section deals with the problems that must be addressed for the formation of a successful reuse reengineering process.

A candidate's reusability can be assessed in a two staged approach. Firstly, it should be considered if the candidate has reuse potential i.e. whether the functionality of the candidate is likely to be required for future development or maintenance. The process of identification and naming of functionality provided by the reuse candidate, is termed the concept assignment process [Biggerst94]. Secondly, the cost of making a candidate reusable should not exceed the overall cost of redeveloping the candidate.

Most approaches towards obtaining reusable software components involve the generation of completely new components and the placement of these components within the library [Lenz87, Cramer91]. There are a number of disadvantages to this approach. Firstly, there is the obvious high cost of the initiation of the reuse library. Secondly, one of the stated advantages of reusing software is said to be its proven quality. It is assumed that reuse candidates will have been well used and therefore the majority of their errors eliminated. With an approach that involves the formation of a completely new reuse library, new component quality is yet to be proven.

The reuse reengineering approach towards obtaining reuse candidates involves the use of legacy systems. The approach broadly involves a search for potential reuse candidates within the legacy code and aims to evaluate the feasibility of separating them from the remainder of the code. Such an approach is much nearer to supporting the quality assumptions than approaches that simply involve the development of new code. Furthermore, the costs of reengineering the reuse candidates to separate them from their original code must fall short of the 'development from scratch' costs if they are to satisfy the reusability criteria.

The reuse reengineering approach also has potential advantages for the maintenance process. If sufficient reuse candidates can be obtained, at reasonable cost, from the existing legacy system, then a code reduction approach can be adopted within the maintenance process. This is the gradual movement to an 'object based' approach. In this approach, the objects are represented by the new reuse candidates that support the old functionality of the legacy system as well as potential new developments. Furthermore, for the new reuse candidates, the reengineering process has improved their interfaces. The availability of well defined interfaces serves to aid the maintenance process by localising changes. This in turn reduces the proportion of the system that will have been updated, by reducing ripple effects, and thus reduces the amount of code that will require, for instance, detailed comprehension.

There are, however, a number of problems with existing reuse reengineering approaches that, if solved, would greatly improve opportunities to identify reuse candidates and to evaluate their reusability. These are as follows:

1. Existing approaches are simply a collection of techniques. Thus, at present, there is no consistent approach for practitioners to follow without previous experience of using a reuse reengineering approach.
2. Many of the essential existing techniques are under-developed, are only usable under certain condition, or are difficult for practitioners to follow and apply.
3. The existing techniques have been developed for many languages. However, no one language has been fully supported and catered for. In some cases, it is unclear for which language the techniques are suited.
4. The results of the techniques are often difficult to interpret and therefore are inaccessible to non-specialist staff.
5. Most of the existing approaches are bottom-up, but the concept assignment process is inherently top-down. This makes giving reuse candidates meaningful descriptions difficult.
6. Individual approaches have only been tested on small sets of code. Typically code modules of 100-1000 lines of code have been used. Thus, it is not possible to assess the suitability of the techniques for industrial applications.

Finding potential solutions to these problems will form the basis of the research described within this thesis.

1.3. Criteria for Success

The criteria for success on which this thesis will be evaluated are the following. The thesis will:

- Describe a method which will effectively collate existing techniques into a number of easy to follow steps.
- Add to, or improve upon, the existing techniques where it proves necessary to support the reuse candidate identification process.
- Provide support for both top-down and bottom-up comprehension processes.
- Provide consistent and complete support for a single language that has been frequently used to implement today's legacy systems.
- Use graphical display of information to support the decision making process necessary for the use of the method.
- Evaluate the results of the use of the method against a number of industrial sized applications.
- Test the method to see if it is supportive of the process of software evolution.
- Indicate the type of tools which should be available to support the use of the method.

The success of the work described in this thesis in meeting these criteria will be examined in Chapters 5 onwards and will be reviewed in Chapter 9.

1.4. Thesis Outline

The remainder of this thesis is organised as follows. Chapter 2 describes the literature that is related to reengineering for reuse. This includes a review of the literature describing the problems of performing maintenance on legacy systems and, in particular, addresses the problems involved with the program comprehension process and reverse engineering. A range of problems are addressed including the lack of, or outdated documentation, information overload when maintaining large systems and the need for information abstraction. A general approach to program comprehension, that of software visualisation, is then described. The ways in which the visualisation of software can aid the program comprehension process is indicated. Finally, Chapter 2 investigates software reuse and proposes the use of a reuse and reengineering approach as a means of gaining acceptance for preventative maintenance.

Chapter 3 reviews existing object identification processes. A number of approaches are described and their benefits discussed. The approaches are grouped into six categories. These include global, type and receiver data persistence approaches. Two additional categories are also defined which include approaches that retain the existing structure of the code as well as more general ones. A review of the general problems that are evident with the use of each approach is presented at the end of each category.

Chapter 4 describes the reuse reengineering approach adopted within this thesis. A ten step method is described. For each step of the method, its objective is indicated together with the tasks to be adopted in order to carry out the step. Small examples to assist the reader are also included.

Chapter 5 defines the requirements for tool support. Some general tools are defined which are required throughout the method. In addition, the needs of some specific tools are defined that are necessary to support specific steps within the method. Current levels of tool support are indicated and problems with their use are described.

Chapter 6 describes a number of case studies that have been carried out in order to test the method. The case studies are from industrial applications, and range considerably in size, age and the extent to which they have been maintained. Examples are selected throughout in order to indicate the usability of the approach. Wherever possible graphical representations are included in order to provide the reader with an indication of the benefits of information abstraction.

Chapter 7 evaluates the method defined within Chapter 4 and tested as documented in Chapter 6. It performs a step-by-step review of the reuse reengineering method. It also identifies deficiencies where further work is required.

Chapter 8 reviews three of the basic approaches of the method. Namely, the use of the dominance tree, data clustering and graphical representations. The review is performed on the use of the dominance tree and clustering by investigating how well they support the evolution process. This provides an indication of the benefits to software maintenance over time.

Within Chapter 9, some conclusions are drawn. The criteria for success are reconsidered and the achievements are discussed. At the end of Chapter 9, several other areas of further work are identified.

Chapter 2. The Related Literature

In Chapter 1 it was indicated how, as software becomes older, and hence has undergone a lot of maintenance, the costs of the maintenance process increases. One approach proposed for dealing with such a problem is the re-development of the software. There are however, a number of problems that are associated with such a re-implementation strategy. These issues are now described.

The strategy of just replacing legacy applications by completely new systems runs the risk of the new system providing differing functionality. The provision of new applications providing similar functionality is not as simple as it may sound. Over their lifetime systems are prone to enhancements and such enhancements are frequently undocumented. Thus, the original application's source code provides the only true definition of the programs functionality. As a means of avoiding the re-implementation of different functionality companies often resort to *reverse engineering* technologies as a solution to their problems.

In order to implement a successful reverse engineering process, the procedure of understanding selected areas of the code needs to be performed. This process of gaining an understanding of the functionality of the code is called *program comprehension*. Program comprehension takes place at many levels; initially a broad understanding of how the program operates may be achieved but this is later followed by a more in-depth understanding of how the system performs specific tasks. The comprehension process assists the reverse engineering process by indicating important aspects of the code such as side effects or anomalies, often by using a number of abstraction techniques such as *software visualisation*. The availability of information gained from the comprehension process is essential if any software migration techniques are to be applied.

As indicated above, performing maintenance is an expensive process. Furthermore, there is a tendency, as maintenance is repeatedly performed, for the maintainability of the systems to degrade [Lehman97]. As a direct consequence, further resources are required to perform successful maintenance on future occasions. One identified approach to improve this situation has been to move towards an object based approach [Canfora94]. The object based approach assumes that individual areas of functionality are grouped into a single object which has a well defined interface. Thus, the communication between objects (hence areas of functionality) is clear and well defined. Maintenance is therefore assumed to be easier to perform as changes are restricted to small numbers of objects.

The necessity to move towards objects leads back to the problems of having to re-implement existing legacy systems. Therefore, excluding the major cost implication of such a large scale re-implementation, there are still the problems of having to re-identify the actual undocumented functionality of the existing systems. If a means can be sought to take the legacy systems and form them into objects, this presents at least a partial solution to the problem. In addition, as full scale re-implementation will not be attempted, costs should be reduced. Furthermore, the objects devised have the potential to be *reused* within the future.

The process of migrating legacy systems into an objected based approach is termed the *reuse reengineering* process, and is the field of study within this thesis. More ad hoc processes are also sometimes termed *software scavenging*. This overview has indicated a number of related fields that feed into the reuse reengineering process. These include *program comprehension*, *software visualisation*, *reverse engineering* and *software reuse*. These are now described in greater detail throughout the remainder of this chapter.

2.1. Program Comprehension

Program comprehension in a software maintenance sense involves acquisition of knowledge about programs, as well as accompanying documentation and operating procedures. Program understanding is the task of building mental models of the underlying software at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for maintenance, evolution and reengineering purposes.

The process of understanding software products involves abstracting information about certain aspects of the software system. During comprehension of a system, the understander tends to form an internal representation, which serves as a working model of the system. At the outset, the model may be incomplete but becomes more complete and accurate as additional information about the system is obtained.

2.1.1 Why Program Comprehension is Necessary

The costs of performing program comprehension have been widely cited as being between 50 and 90 percent of the overall cost of performing maintenance [Standish84]. Furthermore, the costs of performing maintenance has been estimated to be 70 to 90 percent of the total life cycle cost [Standish84]. Assuming these figures are accurate, we can estimate that the costs of performing comprehension for maintenance over the life-time of software could account for approximately 35 to 80 percent of the software life-cycle costs (i.e. development and maintenance costs). Clearly, this is an

important activity so any approach towards assisting the comprehension process can considerably reduce software costs.

Software comprehension activities do not start at a point of a maintainer having no knowledge of a system. Experienced maintainers always bring information to a maintenance project, even if they have no previous knowledge of the software they are maintaining. This knowledge is provided through their previous experiences whether this is experience with the software development language, general software engineering principles or the domain. This previous experience is important for successful maintenance. Thus, it is necessary to support the gaining of further experience to assist program comprehension.

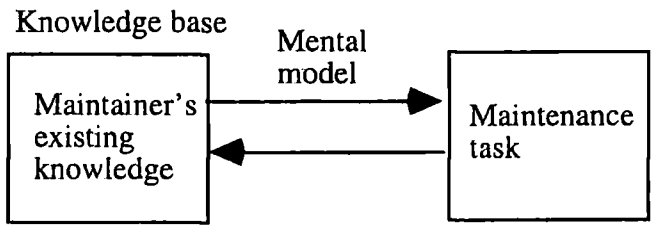


Figure 2.1: Building a Mental Model for Program Comprehension

The maintainers' existing knowledge is referred to as the knowledge base (Figure 2.1). This constitutes all the information to which the maintainer will have access through performing past maintenance activities. Driving the understanding process is the maintenance task. As the maintainer gradually learns about the task in hand, he will begin to devise what is termed a *mental model* of the system and the maintenance task. However, developing an accurate mental model is both difficult and time consuming. Thus, it is the information assimilation process that program comprehension research is trying to support.

2.1.2 Types of Program Comprehension Support

Researchers have defined a number of different types of knowledge that are essential for successful maintenance. The types of knowledge proposed are [Shneider79]:

- **syntactic knowledge**: this represents the type of knowledge which comes from the use of, for instance, programming languages. For example, the key words within a language or the use of a language terminator.
- **semantic knowledge of software engineering**: this knowledge is at a higher level than syntactic knowledge, for instance, it may be a solution to debugging a program or an algorithm to a solve a particular problem.
- **semantic knowledge of task related knowledge**: this may be domain knowledge about how the domain procedures are implemented or how a program operates.

A combination of all types of knowledge is required for successful maintenance.

When gaining information to build up a mental model, a number of different strategies can be applied to the accumulation of information. Types of strategies applied [Littman86] are:

- **systematic strategy**: using this strategy, the entire program is examined to establish the interactions between components. Maintenance begins only once the understanding process is complete i.e. when the maintainer considers the mental model to be accurate.
- **as-needed strategy**: using this strategy the maintainer only attempts to gain an understanding of part of the program. In this case, only a partial model of the software is devised before the modifications process begins.

These strategies can be viewed more simply when related to top-down/bottom-up strategies; top-down being systematic and bottom-up being as-needed. For large software system it is not possible to apply the systematic strategy and, therefore, the as-needed strategy is a more realistic approach for larger applications.

Much discussion by researchers in the field of program comprehension has concentrated on program plans [Letovsky86, Soloway84, Soloway86]. Two levels of program plans are discussed. At the lowest level we have what are termed 'rules of programming discourse'. These rules are said to be composed of the conventions of programming such as the use of meaningful variable names. Thus, the rules of programming discourse are similar to the semantic knowledge of software engineering described above and will be part of the knowledge base of the maintainer. The other types of plans are 'program plans'. These are said to be stereotypic action sequences such as a high level functional description of a loop. Programs are said to consist of a number of plans to fit a specific problem. If plans are well implemented, they can provide the maintainer with a high level of understanding of the program under maintenance.

Unfortunately, over time, plans become de-localised, interleaved, merged or nested with other plans and thus their benefits for maintenance can be reduced. Furthermore, the problem with the information plans provide is that the use of a as-needed strategy with de-localised plans can lead to incorrect assumptions being made about the way a program functions. For this reason, while the as-needed strategy is a more realistic approach to tackling maintenance, it is also a risky process. Both systematic and as-needed strategies need to be applied but at different levels of detail. These would relate more closely to a middle-out approach [Ward94a].

Brooks [Brooks83] investigates a more problematic driven approach to the maintenance task. In his model, the comprehension process is an iterative process of hypothesis verification and modification. In figure 2.1, this process is represented by the arrow pointed from the application task to the

maintainer's knowledge base. He believes that this existing knowledge base provides a set of expectations that are used to derive the initial hypothesis. He calls these hypothesis forming expectations 'beacons'. A plan may be an example of a beacon. The process of hypothesis testing is the way in which the mental model is derived. Further work in this area has also been done [Wiedenbe86, Wiedenbe91] where beacons are viewed as tangible aspects of the program which offer the maintainer distinctive points of recognition and, therefore, support the maintainers orientation through the program. Work in this area has indicated the importance of such beacons but also stresses that carelessly constructed beacons can serve to confuse rather than assist.

2.1.3 The State of the Art for Program Comprehension

There are a number of areas in which the field of program comprehension is progressing. One of these areas is concerned with providing visual representations to assist the comprehension process. However, this area is treated separately in the following section (Section 2.2). Of these areas which do not include visualisation techniques, we consider four to be of greatest importance. These are:

- program slicing
- message sequencing
- plan recognition
- concept assignment

An overview of each of these areas will be presented in the following subsections.

Program Slicing

Slicing algorithms enable programmers to extract the parts of the program that affect a chosen set of variables at some chosen point forming a slice. Slices can be useful in understanding code [Korel98] since they comprise only those parts of the program related to a specific sub-computation (defined by a slicing criterion), while retaining the effect of the original code. Slices can be either static or dynamic. Static slicing contains no information about the execution of the program, and so the slice preserves the effect of the code for every possible execution path. Dynamic slicing is constructed with respect to the traditional slicing criterion together with information on a particular execution pattern such as the input values passed to the program. Dynamic slices yield smaller slices, whereas static slices yield a slice for all possible executions of the program. A combination of both approaches is termed quasi static slicing [Venkates91]. All of these are useful techniques for program comprehension as they serve to reduce the amount of code necessary for comprehension and visualisation.

Work in this area includes understanding function behaviour [DeLucia96], dynamic slicing to understand program execution [Korel97] and amorphous slicing [Harman97] which include a number of simplification procedures to reduce the size of the slice. In addition some research has been carried out on the way the slices of large programs are represented. This includes the representation of slices as pictures [Jackson94].

Message Sequencing

A key problem in building large software systems is understanding the interactions between components. Message sequence charts are a method for understanding interactions between components within complex systems. A message sequence chart displays sequences of messages exchanged between system components within their environment. One implementation of a visual representation of a message sequencing chart is SeeSeq [Eick96]. This system provides three views: an overview, a zoom view and a dictionary view. A number of different display techniques are used which include the use of colour, mouse sensitive selection of detail and variable columns to encode the number of messages sent.

Other approaches include Binder [Binder96] who uses message sequencing as an approach to testing object oriented systems.

Plan Recognition

The recognition of plans can be performed from either a top-down or bottom-up approach or a combination of both. A language independent program representation is used together a translator to transform the source text. A plan recogniser is then used to parse programs by using recognised plans from the clichés held at various levels of abstraction within the library. The recogniser works on the basis of:

- structural information only;
- graph structures;
- similar programs;
- graph grammars and graph recognition algorithms deployed.

The result is a tree or lattice of program components at the leaves and program plans and the goals of the program at the root. An example of the plan recognition approach is Whorf [Steckel92], but more general approaches are based on slicing, including Harman [Harman97] and Ball [Ball94].

Concept Assignment

Concept assignment is the process of discovering human oriented concepts and assigning them to their implementation instances within the software system. For the identification of concepts, simple parsing oriented recognition is insufficient. The human oriented recognition process depends heavily on a priori contextual knowledge of the domain, domain entities and their relationships. Examples of work in the area of concept assignment include Biggerstaff [Biggerst89b, Biggerst94], DeBaud [DeBaud97], Siff [Siff97] and Ning [Ning89].

2.2. Visualisation

Software visualisation is the use of the crafts of typography, graphic design, animation and cinematography with modern human-computer interaction technology to facilitate both the human understanding and effective use of computer software [Price93].

2.2.1 Why Visualisation is Necessary

As indicated above, the various theories of program comprehension put emphasis on the construction of a mental model within the mind of the maintainer. Theories on the construction of mental models hypothesise various techniques employed by maintainers in extracting information from the program source code. Software visualisation attempts to provide tool support for aiding mental model generation. The mental models proposed by various program comprehension researchers are all composed from semantic constructs. These constructs are typically abstractions, at various levels, of program features. The interrelationships identified from these constructs represent the maintainer's understanding of the program.

Software visualisation attempts to provide a mapping from the program code to a visual representation that matches the maintainer's mental model as closely as possible. Highlighting and automatically creating these semantic constructs would reduce the burden from the maintainer, as they would require less time to be spent on scanning the source code. Thus, software visualisation attempts to aid the comprehension process by providing abstractions in a visual form and thereby reducing the information interpretation load. By presenting a pictorial model of complex lower level information, maintainers can rapidly generate an initial mental model of the software and use this as the basis for further investigation. The field of visualisation is therefore considered an important area of research within the field of program comprehension.

2.2.2 Types of Visualisation Support

As visualisation support covers a wide variety of program comprehension tasks, there are a large number of different ways in which to visualise software systems. In particular, we generalise three kinds of display material; program, algorithm and concept visualisations. The concept visualisations represent generic constructs whereas program visualisations represent concrete implementations of code. Algorithms fall midway between concepts and program visualisations. Algorithms are not as concrete as program, since they are generic in that they can be used to implement many programs. However, the term concept refers to a higher level of abstraction than an algorithm as it contains no implementation details.

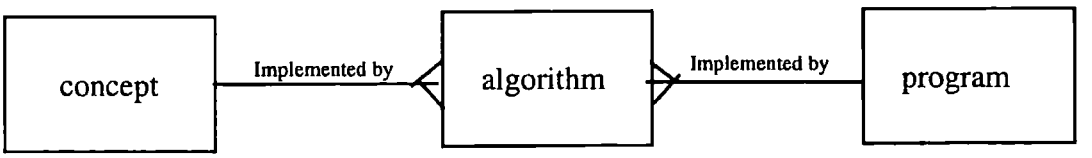


Figure 2.2: Relationships between concepts, algorithms and programs

We are able to derive the relationships shown in Figure 2.2 between concepts, algorithms and programs. Thus, a concept and a program may also be implemented by many different algorithms.

A further difference in the type of information presented is whether static, symbolic or dynamic analysis is used to form the visualisation. These terms represent the way in which the data for visualisation is collected; dynamic analysis involves the execution of the source code that is not required for static analysis but symbolic analysis is a combination of both approaches. It is possible to investigate the static, symbolic and dynamic nature of algorithms [Brown88] and programs, but not for concepts due to their lack of implementation details.

A graph showing the relationships between the visualisation approaches is shown in Figure 2.3.

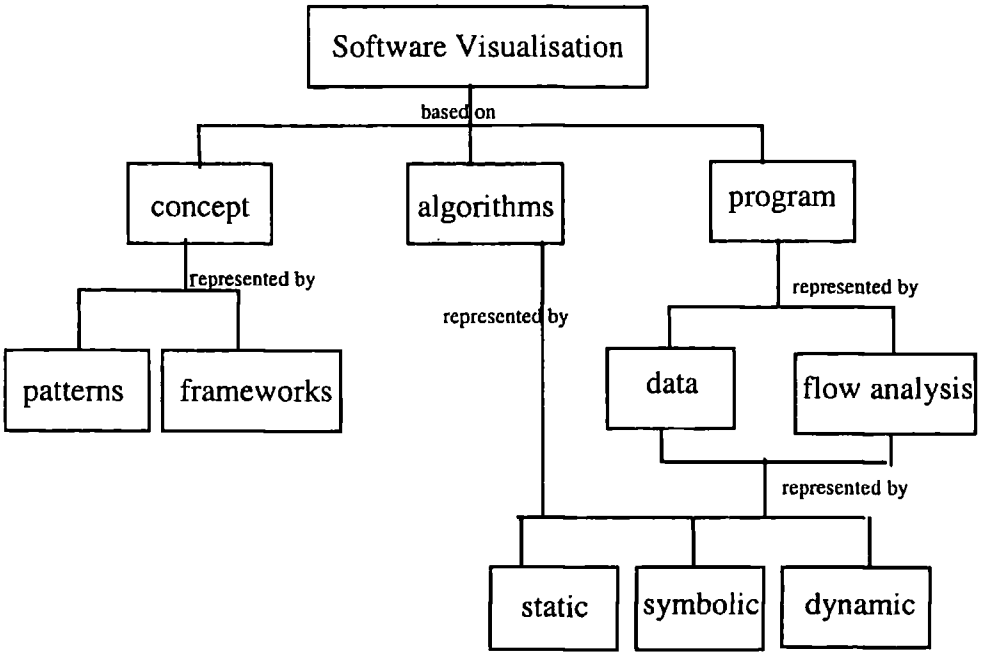


Figure 2.3: Types of visualisation support

The visualisation of concepts usually forms a central part of the design of object oriented systems. In this case generic patterns and frameworks and used to represent general concepts within software development such as abstract and reusable object groupings. The visual representation of these concepts is usually a set of object represented in an object design notation. Examples include Pree [Pree95], Gamma [Gamma95], Bassett [Bassett97] and Fowler [Fowler97].

2.2.3 The State of the Art for Software Visualisation

Over the past couple of years software visualisation research has been directed towards structural or macroscopic visualisation of code. Concentrating, in particular, on a structural view of code typically producing call graphs, control flow graphs and file dependencies. In these examples variables, classes, functions methods, files and modules are displayed to produce the stereotypical views. The key factor within these views is their inherent graph structure. The disadvantage with these graphs is that they are often very large, especially for industrial sized applications, and so they are difficult to display easily. Examples of these systems include Logiscope [Meekel88], Rigi [Storey95], Narcissus [Hendley95], GraphLog [Consens92], SemNet [Fairchil88] and GraphTool [Bodhuin94].

Algorithm visualisation research is geared towards understanding the way in which the algorithm operates. Generally, these are not produced automatically and require a new representation for each algorithm. Furthermore, many algorithm animation frameworks rely on the instrumentation to place additional calls to the animation environment or to a log file generator to drive the animation. Further details of these systems are described within [Price93, Myers90, Roman93].

There are a large number of systems that use visualisation as the primary aid to debugging. The majority of these systems are concerned with visual representation of traces of execution paths. These systems are mainly automatic and operate either through an interpreter base, where the target program is executed, or the visualiser augment the target source code with calls to the visualiser.

Several programming environments have been developed which include visualisation capabilities. PECAN [Reiss85] and PROVIDE [Moher88] are early development systems that utilise graphical views for the display of data structures, call-graphs and call stacks. FIELD[Reiss90] contains an extensive set of tools for developing and maintaining C++ programs which include class browsers [Lejter92] and flow graphs and more recently 3D display of views [Reiss93]. The BEE++ [Bruegge93] object oriented application framework supports dynamic analysis of distributed programs.

Much of the research investigating the presentation of object oriented programs has focused on the graphics used to aid the design process. The work of Booch [Booch91], Rumbaugh [Rumbaugh91], Beck and Cunningham [Beck89] and Coleman et al [Coleman92] all are notable leaders within this area.

The majority of the visualisations systems to date have concentrated on producing 2D representations. A growing area of research is investigating the application of 3D graphics for software visualisation. There is an increasing need for more expressive powers to represent the inherently multi-dimensional [Brooks87] data, facts and relationships that constitute a software system. Researchers in the 3D field have attempted to address the representation and navigation issues. As with 2D representation the problems with increasing complexity for representing large applications are still prevalent. While 3D technology cannot immediately solve this problem, it does offer a number of advantages such as greater working volume for information presentation. These arguments do not in themselves justify 3D as superior to 2D representation, however they do highlight possibilities for advances in software visualisation using 3D graphics. Examples of these systems include Zeus [Brown93], PVMTrace [PVMTrace95], VisualLinda [Koike95], SemNet [Fairchil88], GraphVisualizer3D [Ware93, GV3D95] and Narcissus [Hendley95].

2.3. Reverse Engineering

Although there are a great number of definitions for reverse engineering the definition by Chikofsky and Cross [Chikofsk90] is the commonly accepted one. Their definition of reverse engineering is:

> *the process of analysis of a subject system to:*
> - *identify the system's components and their interrelationships and*
> - *create representation of the system in another form or at higher level of abstraction.*

The term system's components refers to the products from different phases of the software life-cycle, for example, the specification, design or the source code. Although the reverse engineering process can start from any of these components, the most common starting point is the source code. This is because it is often the only available or accurate description of the systems functionality.

2.3.1 The Benefits of Performing Reverse Engineering

The goal of reverse engineering is to facilitate change by allowing a software system to be understood in terms of its functionality as well as its current implementation approach. Furthermore, an effort is made to recover lost information to assist maintenance, aid migration, reduce side effects and / or extract reusable components.

There are a number of different approaches to reverse engineering software projects. These are:

- redocumentation
- design / specification recovery

These are now described in further detail.

Redocumentation

Redocumentation is the recreation of an equivalent representation within the same level of abstraction [Chikofsk90]. There are three main goals in performing this process. These are to:

- create alternative views of the system, usually from the source code, so as to enhance understanding of how the system works.
- improve current documentation by either updating the present documentation or replacing absent documentation.
- generate documentation for a newly modified program in order to facilitate future maintenance.

Examples of redocumentation projects include Wong [Wong95] and Ahrens [Ahrens95].

Design / Specification Recovery

Design recovery entails identifying and extracting meaningful higher level abstractions beyond those obtained directly from examination of the source code [Chikofsk90]. Unlike redocumentation, which is

usually formed from the source code design recovery also requires far more personnel experience and knowledge of the application domain.

A number of different approaches have been used for the recovery of design information. One approach by Rugaber et al [Rugaber90] relies heavily upon programming language constructs and is based on the belief that design recovery is being able to recognise, understand and represent design decisions present in the source code. Montes de Oca supports a similar approach with data mining techniques [Montes98]. Other approaches rely on the knowledge about the problem and application domains in general. For instance, the Programmer's Apprentice [Rich90] offers support for 'automatic cliché' recognition. A cliché is a commonly used combination of elements [Wiedenbe91] and assumes that once the cliché has been identified an experienced programmer will be able to reconstruct the appropriate designs.

Specification recovery involves identifying, abstracting and representing meaningful higher level abstractions beyond those obtained by inspecting the design or source code. The approach generally adopted involves the derivation of the specification through backwards transformations. Ideally the resulting specification should be represented in a form that can be easily re-implemented within another programming language. Thus reducing the potential 'mistakes' that can be made in the specification to code translation process. Usually it is aimed to retain the precise semantics of the code. Examples include Lano and Haughton [Lano94], and Ward[Ward94b, Ward95].

The specifications recovered can also be in the form of 'object classes'. This approach is particularly useful if the final destination of the reverse engineering process is to be an object based approach.

The benefits of the recovery of specification and design information are:

- they can be used to support software maintenance without the necessity to constantly refer to the low level source code.
- they assist maintainers to more quickly acquire the appropriate level of understanding required to effect changes to a system.
- they can be used in the future implementation of similar systems.

There are a number of techniques used within reverse engineering which help to recover the lost information. These include:

- re-structuring
- reengineering

These will now be described in more detail.

Re-structuring

Restructuring is the process of transforming a system from one representational form to another but without altering the level of abstraction. The aim of the transformation process is to produce a more desirable format. Therefore, restructuring can be seen as a form of preventative maintenance.

There are various types of restructuring that differ in the part of the system affected and the way in which the restructuring is performed. These include:

- Control flow driven restructuring - This process involves the imposition of a clear control structure on the source code.
- Efficiency-driven restructuring - This process involves restructuring a function or algorithm to improve its operating speeds.
- Adaptation driven restructuring - This process involves changing the coding style in order to adapt the program for a new language or operating environment.

In addition to source code, other representations can also be restructured. These include requirement specification, data models and designs. Although it is difficult to completely automate the restructuring process, due to the need for human intervention, there are a number of semi-automated tools [Griswold93].

Reengineering

Reengineering is the process of examining and altering a target system to implement a desired modification. The overall reengineering process consists of two steps. Firstly, reverse engineering is applied to the target system to understand and represent it in a new form. Secondly, forward engineering is applied to implement and integrate the new aspects of the system.

The Colbrook et al [Colbrook90] Source Code Reengineering Model breaks this two staged process into further stages. Those that represent the reverse engineering process are encapsulation, transformation, normalisation, interpretation and abstraction. Those that represent the forward engineering process are causation, regeneration and certification.

2.3.2 Reverse Engineering Methods

In the subsections above the approaches to reverse engineering are discussed. In the following subsections some of the methods for reverse engineering are described.

RECAST

The RECAST method [Edwards93, Edwards96] provides a route for reverse engineering legacy systems into SSADM logical specifications. It considers all the code relevant to the system as well as the user interface. Furthermore, the logical specification is derived from the data, processing and on-line aspects of the system. It is assumed by RECAST that it is possible to identify all the relevant components of the source system that is to be reverse engineered. Therefore the following elements must be available:

- the Job Control Language (JCL) of the run units
- the original COBOL source code (before any pre-processing is carried out)
- the form of any copy code extracted from copy libraries
- the form of any code held in the data dictionary

RECAST extracts a no-loss representation of the current software system and documents the resulting system specification in the form of SSADM diagrams and notations.

The RECAST method is outlined by a structural model. This model defines the inputs and outputs of each of the stages within the method. The structural model has four stages:

- Identification of business users' views (BUV)
- Identification of the logical data model (LDM)
- Identification of the system processing (SP)
- Identification of the menus and dialogues (MD)

The dependencies between the stages and the structural model are shown in Figure 2.4.

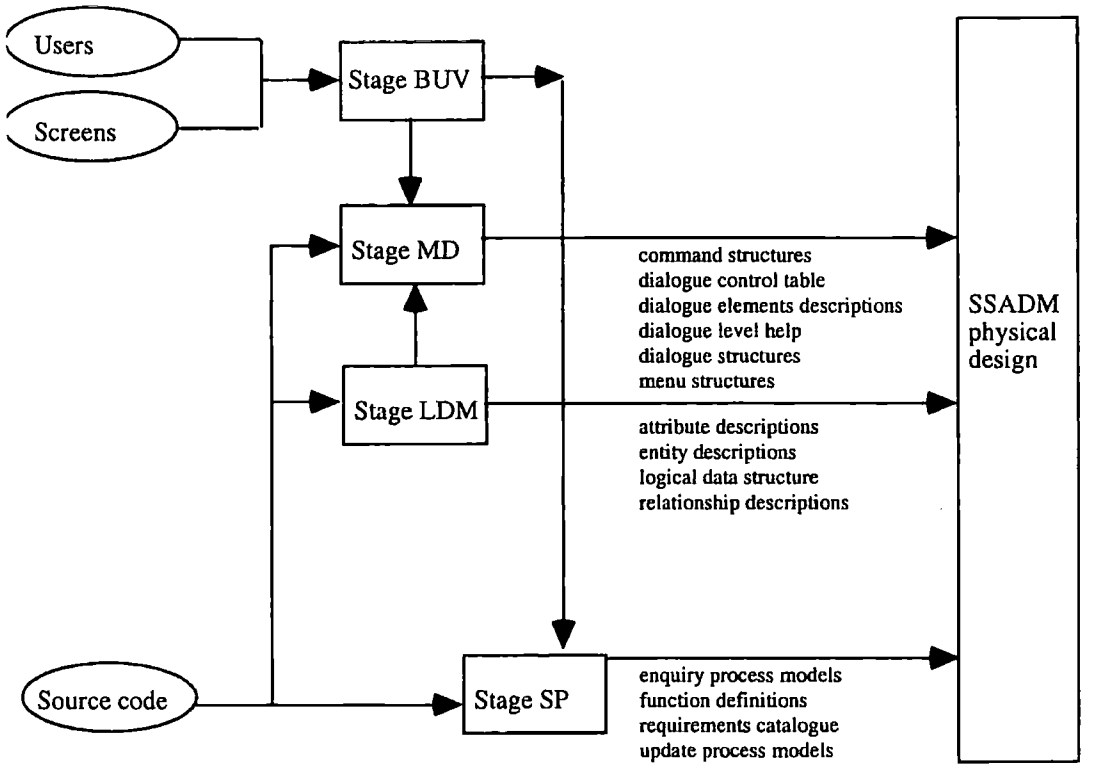


Figure 2.4: The Dependencies between the RECAST stages

The identification of business users' views is based on information gained from the business user domain whereas all other details are based on information gained from the existing source code. The stages are now described in detail.

Identification of Business Users' Views

This stage acquires information from the business users of the system. It considers the inputs and outputs of the system and identifies what the users' perceive as the function of the system. It does not consider the system's source code.

Identification of the Logical Data Model

This stage has six steps. The first three steps define the rules for analysing the data. Steps four and five then define the rules for dealing with the schemas and subschemas if an IDMSX database is accessed. The final step draws together the two strands into one full outline data model.

Identification of the System Processing

This stage of RECAST identifies a set of system processes, collectively termed a function, which the users wish to schedule together along with their component event / enquiry level processing. This stage also restructures the functions into logical subsystems.

Identification of the Menus and Dialogues

This stage is only carried out for those systems that have some element of on line processing. In addition, its rules are specific to the ICL TPMS transaction processor.

REDO

The REDO project [vanZuyle93, Lano94] is a combination of techniques into prescriptive methods which can be applied to any existing application whatever its state. The results of its application will produce a reengineered, renovated, reconstructed or reverse engineered application. The new application will have the same or very similar functionality, but improved maintainability. The method is believed to be very generic and can be customised to specific circumstances. The objectives of the REDO method are that the method should be up-to-date, practical to use, comprehensive but simple to follow, based on systematic, incremental and established techniques producing reliable results. Furthermore, it should be independent of any maintenance environment and supportable by automatic tools.

The generic method is as follows:

Stage 1: Assess existing state
This stage covers the gathering, recording, classification and assessment of existing sources of information, including design documents, sources files and configuration management records. The result of performing this step is an archive of all the relevant items identified and an analysis of the current state of the application.

Stage 2: Install application in Reengineering Environment
A suitable reengineering environment must be designed, acquired and set-up. This will include a variety of facilities, including editing facilities, for generation of documentation and configuration management. The baseline chosen in stage 1 must be imported and upgraded to be maintainable during the reengineering process in the new environment. The environment and its contents constitute the IRSE (Integrated Reengineering Support Environment).

Stage 3: Reverse Engineer

Stage 3 takes the existing information recorded in the IRSE during stage 2, analyses it and merges in a range of other information, in order to complete the documentation package. The package may include not only traditional documents but also on-line query and hypertext facilities. Thus, stage 3 produces whatever documentation package is required within the constraints of the data, knowledge and resources available. The information is returned to the IRSE for possible further refinement in later stages.

Stage 4: Establish Test Procedures

Stage 4 establishes an appropriate test procedure for the application, incorporating any suitable existing material. It follows Stage 3, since most forms of testing presuppose a specification of expected behaviour. If reengineering or re-implementation is to be attempted, the creation of any required implementation-specific tests should be included as part of those activities.

Stage 5: Reengineering / Re-implement

Stage 5 is concerned with improving the implementation. Thus any change to an application's source code will be performed in this step to improve its comprehensibility. Reengineering and re-implementation are treated as one process because they are extremes and thus require no precise distinction in practice. This stage includes the preparation of any required implementation-specific tests. Previous implementation specific tests may well be helpful. Ideally, these would be upgraded automatically as the implementation is changed / replaced.

Stage 6: Handover

This is the final stage of the REDO method which installs the definitive version of the application in operational use through whatever release control mechanisms are applied by the organisation and transfers responsibility for it back to the maintainers.

2.4. Software Reuse

There are many differing definitions of software reuse. These vary with the approach to reuse taken and the types of products that are considered. Recently, the definition used within the REBOOT project has begun to be accepted as the industry standard. This definition originates from Krueger [Krueger92]:

> *'Software reuse is a process of creating software systems from existing software assets, rather than building software systems from scratch'.*

This section will consider the benefits that software reuse can provide, as well as some of the difficulties putting reuse into practice.

2.4.1. The Promises of Reuse

The goal of a software development organisation is to build useful working software systems in time and within given economic constraints [Krueger92]. Reuse has the potential to assist the process of achieving this goal. The essence of software reuse is to use any information, artefact or product held in the software company's inventory. The reusable element can be code, requirement or design specifications, or knowledge of the domain. It can also include processes, methods and templates that are known to be effective in specific cases.

Many varying sets of statistics are available on the potential degrees of reuse that can be achieved. Langergan [Langerga84] has indicated that as much as 60% of the code in data processing applications is redeveloped unnecessarily. Jones goes further than this and states that less than 15% of new code serves an original purpose [Jones84]. While figures may vary as to what levels of reuse should, or could, potentially be achieved, what does not vary is the view that reuse is viable. Indeed, in Japan, such principles have long been accepted.

Myers [Myers85] investigating differences between the US and Japanese software industries found that:

> *'The US industry standard for productivity is approximately 3,600 source code lines per year. In Japan, where the reuse of reusable software is more widely practised, the industry standard is approximately 24,000'*

Other studies have indicated similarly high results and, in some studies, Japanese software factories have been shown to have reuse factors of 85% [McNamara84]. Therefore it is believed that reuse is critical if we are to deliver efficient, reliable and maintainable software on time [Frakes88]. If Western software production is to be able to compete with the Japanese then it seems that the inclusion of reuse in the software development process is necessary but this will require a major change in attitude from the software industry [Lubars88].

Increased productivity is not the only benefit that reuse offers. Agresti [Agrest88] believes that, apart from the productivity gains, the benefits of reuse are:

- Reliability - this is achieved by the use of proven components
- Consistency - this is achieved by using the same components in many places therefore reducing the need for fresh and possibly idiosyncratic design

- Manageability - achieved through the use of well-understood components as reuse reduces the likelihood of cost and schedule overruns by providing already developed components whose behaviour is understood
- Standardisation - through the use of standard components

For software to be reusable it must exhibit a number of features that directly encourage its use in similar situations. The following findings by Wood [Wood89] are considered to be the important qualities of reusable software.

- 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
- Reliability - components should be error free
- Portability - components should not be restricted in terms of the hardware or software environments in which they operate

It should be noted that the above are not prerequisites for reuse and that many may conflict, but, the more points that are satisfied, the more feasible the practice of reuse becomes. Many of these points relate to the 'packaging' that surrounds a reuse component and therefore, indirectly, the operations that the unit performs and its links to other components. It is possible to additionally use this above list to begin to identify a number of problems associated with the reuse of software components. These problems are investigated in more detail in the proceeding section.

2.4.2 The Problems Associated with Reuse

The difficulties that are concerned with the implementation of reuse involve both managerial and technical issues as well as a number of other aspects of development such as legal issues. Meyer [Meyer87], investigating why reuse is not more common, concentrated particularly on the managerial aspects of the problem. He considered these to include:

- the fact that economic incentives tend to work against reuse
- the 'not invented here' syndrome
- the problem that designing for reuse is hard

- the fact that requirements vary over time and from project to project etc.

Cramer [Cramer91] et al, investigate the technical problems associated with reuse. These are:

- Development - development process of reuse aims at populating a library with reusable software components. This area includes the software component model representing the principles and concepts the software components have to adhere to. Wilkerson [Wilkerso90] has carried out work in this area. Notations, i.e. the formalisms, used for describing software components according to the software component model. Cheatham [Cheatham84] has carried out work in this area. The development model, i.e. the way in which the notations are used to describe the software component. The development tools, i.e. the software that supports the development of the reusable software component. These include knowledge capture [Latour88] transformation tools [Reps84, Spicer91, Reubenst91] and evaluation procedures [Cardenas91].
- Certification - the process where written guarantees must be given identifying that a component complies with its specific requirements and is acceptable for operational use [Glaser90]. The quality requirements for making a software component re-usable need to be defined more strongly, than they are for non-reusable components. They must also be more strictly controlled. Quality matrices are commonly used as a way of providing a solution to this problem [Basili90, Troy81].
- Classification - in order to be able to locate components some means of being able to sort components logically into groups must be sought. One popular classification mechanism is the faceted approach that was originally developed by Ranganathan [Ranganat91] in the late 60s and then enhanced for use with reusable components by Prieto-Diaz. This method has been successfully used in a number of projects [Prieto-D87b, Dineur89].
- Retrieval - retrieval of components is the process of identifying the most appropriate software component by matching certain functional and non-functional requirements deduced from the prospective system [Cramer91].
- Modification - any differences that are highlighted by the evaluation process between the prospective system and the units that are to be reused need to be recorded so the appropriate modification to the units can be made. The majority of the work in this area has concentrated on the reuse of code. An example of other solutions in this area is the use of parameterization [Goguen84]. Modification is somewhat entangled with the necessity for cost predictions for the necessary changes. For the reuse of components to be feasible then the advantages gained for the reuse of the unit must be greater than the cost of redeveloping the unit. Studies have found that people tend to underestimate the amount of effort needed to modify a candidate [Woodfield87]. Work in this area includes [Conn88, Gaffney88, Barns91].
- Integration - is the process of including the reuse units into the prospective system. Thus there is a need for good documentation and well-defined component interfaces. Object

orientation is one approach to this problem, others include 'Module Interconnection Languages' [Prieto-D87a], adaptive interfaces [Purtilo91] or simply the design of general methodologies for integration [Kang88].

Hall [Hall88] adds to this list of technical problems by identifying the further following key issues. These are:

- Identification - identification of software elements as candidates for reuse, thus finding those candidates suitable for reuse. Canfora, Cimitile and Visaggio [Canfora95, Cimitile95] have performed work in this area.
- Abstraction - the principle of ignoring those aspects of a subject that are not relevant to the current purpose to concentrate solely on those that are [Glaser90]. Work has been carried out in this area by many of the researchers working in the field of object orientation [Morrison87, Snyder86].
- Cataloguing/storage - these are extensions of the classification problem and include issues of where software components are physically located.

However, Freeman [Freeman84] points out that besides the more traditional technical and managerial requirements for successful reuse there are a number of other points that must be considered. These include:

- the standardisation of the life-cycle and the work products
- the education of both technical people and managers.

2.4.3 Implementing Reuse

The experience of implementing reuse in the software industry has mainly concentrated on the reuse of code. This has been implemented in a number of ways. These are:

- Composition based systems - The principles behind composition based systems is that components suitable for reuse are collected together in a library. When reuse occurs, suitable units are taken from the library and are integrated into the new development. Examples of the use of this approach include Meld [Kaiser87] and [Tracz87].
- Generation based systems - In this case what is being reused is the generation process. Biggerstaff [Biggerst89a] distinguish some subclasses of this approach: language based systems; application generators; and transformation-based systems. Language based systems are those in which the specification language is well defined so that it is able to represent adequately the problem domain and hides the details of implementation from its user, for instance SETL [Dubinsky89]. Application generators embed in their design the

architectural pattern that will be reused in the course of generating specific instances of target systems, for instance, Draco [Freeman87]. Transformation Based Systems focus on the role, structure and operations of transformations in the elaboration of high-level specifications into fully functional programs. Work in this area includes Cheatham [Cheatham84] and Ward [Ward94b, Ward95].

- Pattern based approaches - The pattern approach uses a combination of the component and generation based ideas. The design pattern concept can be viewed as a set of rules describing how to accomplish certain tasks within software development. According to Coad [Coad92], observing the lowest level building blocks (classes and objects) and the relationships established between them identifies design patterns. Patterns are available for a number of stages of the lifecycle. For example see Vlissides [Vlisside96], Pree [Pree95] and Fowler [Fowler97]

Each of these different approaches to reuse will have a different set of technical problems with which it is associated.

2.5. Summary

This chapter has looked at the areas of research related to reuse reengineering. In particular, it has considered program comprehension, software visualisation, reverse engineering and software reuse. An overview of some of the research in each area has been presented.

Chapter 3. Object Recovery

Many programmers adopt an ad hoc approach to obtaining reusable software by scavenging fragments from existing software systems and reusing them. Code scavenging is used to reduce the cognitive effort and the amount of time required to design, implement and debug a new software system. The code is then specialised by manual editing which attempts to resolve differences between the original context and its new context. The programmer must therefore understand the lowest-level details in the code fragment to adapt it to its new context. In addition, further modifications may be required to be performed to the scavenged code or the new system to insure that integration is successful. This often means that the expected reduction in cognitive effort and development time is not achieved.

The scavenging process has the potential, when used more formally, to reduce the effort required to generate reuse components. In addition, as indicated in Chapter 1, this process, when used with reengineering, is also a form of preventative maintenance.

According to Lui and Wilde [Lui90], gaining knowledge as to what objects are within a program are important to:

1. Understand the systems design more precisely.
2. Facilitate reuse of existing methods contained within other systems.
3. Avoiding degrading the existing system design by introducing unnecessary references to data that should be private to a given class of objects.
4. Reengineer the system from a conventional programming language to facilitate future enhancements.

A number of projects have attempted to provide much needed support for the scavenging process. Livadas and Johnson [Livadas94] have grouped object identification projects into three main types of approaches. These are global, type and receiver based. However, when grouping existing work concerned with object identification, some approaches do not fit into these categories. Some approaches use data persistence as a grouping method, others use the existing modular structure. In addition, some methods will use more then a single approach to identify new objects. Thus, in this thesis, 6 different categories are used to represent the object identification process. These are:

- Global based - global based object identification defines a candidate object to be a triple (ϕ, Ø, x) where $\phi \subset F$ (F = a set of routines) and x is a global variable that is visible to each element of ϕ.
- Type based - type based object identification clusters a routine with the set of types of all its formal parameters and the type of its return value.
- Receiver based - receiver based object identification clusters its routines with the type of its input parameters (its receivers).
- Data persistence - this approach groups components on the basis of the data files written by the code.
- Existing structure - the existing modular structure is translated directly into new objects.
- General approaches - these methods use a selection of some of the approaches described above as well as provide a set of guidelines as to how the objects should be grouped.

While each of these categories describes a distinct approach to the identification of objects (with the exception of the general approach), many of the methods may overlap. This overlap has resulted from attempts to improve the object identification process. Such a result would seem to indicate that single category solutions are in themselves inadequate. Descriptions and examples of approaches for each approximate category are given in the proceeding section.

3.1. Global Based Approaches

A global based object identification approach defines a candidate object to be a triple (ϕ, Ø, x) where $\phi \subset F$ (F = a set of routines) and x is a global variable that is visible to each element of ϕ.

3.1.1 The Lui and Wilde Method 1 Global Based Object Finder

Lui and Wilde [Lui90] have found two broad methods useful for identifying objects from C code. The first approach is based on global and static data and establishes links to the routines that manipulate such data.

The steps for this method are as follows:

1. For each global variable x (i.e. a variable shared by a least two routines), let P(x) be a set of routines which directly use x.
2. Considering each P(x) as a node, construct a graph G = (V,E) such that

$V = \{P(x) \mid x \text{ is shared by a least two routines}\}$

$E = \{P(x_1)P(x_2) \mid P(X_1) \cap P(x_2) \neq \emptyset\}$.

3. Construct a candidate object (F, T, D) from each strongly connected component c= (Vc, Ec) in G such that

$$F = \bigcup_{P(x) \in Vc} P(x)$$

$$T = \emptyset$$

$$D = \bigcup_{P(x) \in Vc} \{x\}$$

However, this method in many cases may produce objects that are too big, since any routine that uses global data from two objects creates a link between them. Thus, a further stage of refinement will probably be necessary in which human intervention or heuristically guided search procedures improve the candidate objects by excluding offending routines or data items from the graph G.

3.1.2 The Achee and Carver Approach

This method [Achee94] analyses the program at the subprogram level and uses a data-driven, bottom-up approach to construct objects. An object O, is defined as a two-tuple (D,M) where D is the set of data items and M is a set of methods that act on those data items. The bottom-up approach constructs objects by determining the cohesive strength between two data items.

By considering each subroutine as a unit of functionality, the actual parameters are those necessary to perform the function of the given subroutine. Based on these guidelines, the algorithm seeks to obtain the smallest set of parameters needed for the strongest cohesive unit. The cohesive strength of a pair of parameters is measured by determining the frequency in which they are both necessary to perform various functions.

The approach taken results in a costing function that, for a given pair of parameters, weights the necessity of both parameters of the pair as a stronger condition than the necessity of only one parameter of the pair. Thus, the cost function for a pair of parameters i and j with respect to subroutine f is as follows:

c(i, j)	= c(i,j)	iff neither i nor j is necessary for the execution of f
	= c(i,j) + .2	iff both i and j are necessary for the execution of f
	= c(i,j) - .1	iff either i or j (not both) is necessary for the execution of f

Objects are determined by first grouping the data to form attribute sets and then affixing methods to the sets of attributes.

3.1.3 Overview of Global Based Approaches

The benefit of applying the global based approach will depend upon the language that is used to implement the legacy system. Livadas and Johnson [Livadas94] have identified a number of problems with using this approach with Pascal. They have identified that with nested procedures, non-local variables can be regarded as global for the purpose of the object finder. Secondly there are also problems with the algorithms for the Liu and Wilde [Liu90] global variable usage due to actual parameter passing. For instance, if a variable x is global to f1, and f1 calls f2 with x as an actual parameter, the interpretation should be that x is global to f2.

Further problems with the global based approach can be identified with COBOL. Since within the COBOL language all data items are global within the code module the connectivity between each and every data item is likely to be very high. This would therefore seem to indicate that for COBOL legacy systems the approach is impractical.

3.2. Type Based Approaches

A type based object identification approach according to Liu and Wilde [Lui90] clusters a routine with the set of types of all its formal parameters and the type of its return value. The definition provided by Livadas and Johnson [Livadas94] excludes consideration of return types to distinguish it from their receiver based approach (Section 3.3).

3.2.1 The Lui and Wilde Method 2 - Types based object Finder

Lui and Wilde [Lui90] define two broad methods useful for identifying objects from C code. The first approach has been described above, the second is based upon data types. This approach establishes relationships between types and the routines that use them for formal parameters or return values. The method is as follows:

1. Ordering - define a topological order of all the types in the program. This is achieved as follows:

a) If type x is used to define type y, then we say x is a sub-type of y and y is a super-type of x, denoted by x << y

b) If x << y and y << z then x << z (transitivity).

2. Initial classification - construct a relationship matrix R(F, T) in which rows are routines and columns are types of formal parameters and return values. Initially, all entries of R(F, T) are zeros. An entry R(f, t) is set to 1 of type t is a subtype of the type of a formal parameter or of a return value of routine f.
3. Classification readjustment - for each type, a super-type is considered to dominate the role of classification. Therefore, for each row f of the matrix R, mark R(f, t) as 0 if there exists any other super-type of t on the same row which has been marked with a 1.
4. Grouping - collect the routines into maximal groups based on sharing of types. Specifically, routines f1 and f2 are in the same group if there exists a type t such that R(f1,t) = R(f2, t) = 1.
5. Construct a candidate object (F,T,D) for each group where:

   F = {f | the routine f is a member of the group}

   T = {t | R(f, t) = 1 for some f in F}

   D = Ø

The process of finding objects is then as follows:

1. Let the user propose a starting point - one or more routines or data items to be part of an initial candidate object (F, T, D).
2. Let the user choose which of the next to supply: data-routine analysis, routine-data analysis, type-routine, or routine-type analysis:
   a. If data-routine analysis is chosen: add to F the set of all routines that use data items in D.
   b. If routine-data analysis is chosen: add to D the set of all global and persistent data items used by routines in F.
   c. If type-routine analysis is chosen: add to F the set of all routines that use types in T for formal parameters or return values.
   d. If routine-type analysis is chosen: add to T the set of all types used as formal parameters or return values by the routine in F.
3. Draw a graph showing the F, T, D and their connections based on global and types analysis (methods 1 and 2) for review. The user may delete any of the components, repeat the previous step, or store the candidate object for future reference.

Further work in this area has been carried out by Deursen [Deursen98] who investigates gaining type inferences from languages that have a limited type system. His approach ensures the improved coverage of development languages for the use of the approach described above.

3.2.2 Direct Slicing

Cutillo, Fiore and Visaggio [Cutillo93] use a modified version of Weiser's slice [Weiser84]. This they term direct slicing. This form of slicing makes it possible to separate and extract particular kinds of code segments, distributed among the modules of a program. This method is detailed below:

1. Analyse the program for data and control flow.
2. Find and save all, and only, those nodes handling structures of certain kinds (e.g. a record of a file).
3. For each different data structure, produce a subset of the sets in point 2 containing those nodes which manage a single data structure of the predefined type.
4. For each of the nodes of the subset obtained in step 3:
    a. Find all the nodes that directly or indirectly modify the variables at the considered node.
    b. Save the nodes found in 'a'.
    c. Flow backward from the considered node to each node contained in the saved set. (This identifies the paths that lead from the considered node back to each node within the saved set.)
    d. Find in the results of 'c' all the nodes that directly modify the variables at the considered node.
    e. Find in the results of 'c' all the nodes that manage the flow of control. (These are conditional nodes and / or jump instructions.)
    f. Merge the results of 'd' and 'e' into a set of nodes to constitute the slice.
    g. If the result is acceptable, extract the resulting slice together with the data declarations needed to compile the resulting module correctly.

3.2.3 Formal Methods

Lano, Breuer and Haughton [Lano93] describe how they aim to create object-based abstractions to capture design and functionality. Central to the process is tool support for transformation between the stages. The approach adopted is to transform the COBOL to an intermediate language called Uniform. The Uniform is then translated into a functional description language and finally to the Z specification language. During this process, dataflow diagrams, entity relationship diagrams, call graphs and other types of information are extracted from the code.

The three main stages of this process are:

1. Clean - translation to the intermediate language Uniform, eliminating redundant language constructs. Also, relationships between data values (such as REDEFINES) are translated

into statements about invarients in the program's run-time behaviour. The clean stage can be seen as restricting the original language to a small subset of permissible constructs because Uniform corresponds semantically to a restricted subset of COBOL.

2. Specify - Using dataflow diagrams for guidance, associated variables are grouped together to create prototype objects. These are object based entity descriptions consisting of lines of attributes and their types, perhaps with initial values, but as yet containing no list of associated operators.

   The code is then split into phases - maximal logically connected sections of code within which no files are opened or closed, or have their read / write status changed. Equational descriptions of the functionality associated with these phases are obtained automatically and transcribed into the intermediate functional language, simplifying transforms being automatically applied in order to reduce the equational presentation to a normal form.
3. Simplify - The abstracted functional descriptions are incorporated into the outline objects as descriptions of their operations, thus filling in the semantics of the prototype objects identified during stage 1. A full specification (in Z++ [Lano91]) is then printed using the object based abstraction as a basis, along with associated textual documentation.

Many aspects of these stages are automatic; however, some processes necessarily require interaction with the maintainer, which allows his intuitions and domain knowledge to be utilised in producing reverse-engineered abstractions.

3.2.4 Overview of Type Based Approaches

The type based object identification process clusters routines with the set of types of all its formal parameters and the type of its return value. Livadas and Johnson [Livadas94] have observed through the use of this approach that, in some cases, the routines should not be clustered with their return value as the approach sometimes produces misleading clusters. One cited example is a grouping based on the use of the boolean return value.

The type based approach also assumes that there are types defined within the code to be analysed or are defined within the language definition. This is not the case with COBOL as there is no parameter passing and no true typing. With this regard the approach is limited by the necessity to make assumptions based on the GROUP and ELEMENTARY items and their PICTURE representations which defines the storage limitations of the data items.

Unfortunately, approaches like direct slicing require considerable experience to identify 'acceptable' results.

3.3. Receiver Based Approaches

A receiver based approach clusters its routines with the type of its receiver. A receiver parameter type (or simply a receiver) of a routine f is the type of a parameter that is modified in at least one execution path of f.

3.3.1 OBAD's Recovery of Abstract Data Types

Harris, Reubenstein and Yeh [Harris95] worked on some of the early ideas for OBAD, specifically investigating reverse engineering at the architectural level. Their work has led to the definition of an architecture recovery framework. This framework is made up of three components:

- an architectural representation that supports both idealised and as-built architectural representations with a supporting library of architectural styles and style components.
- a source code recognition engine and a supporting library of recognition queries.
- a 'Bird's Eye' program overview capability

The framework supports architectural recovery in both a bottom-up and top-down fashion. In bottom-up recovery, analysts use the Bird's Eye view to display the overall file structure and file components of the system.

Within the source code recognition engine and library of recognition queries are data abstraction queries. The recogniser is used to recover abstract data types. The recogniser is called OBAD[Yeh95]. When recovering abstract data types OBAD assumes that an abstract data type is implemented as one or a few structure types whose internal fields are referenced by the procedures that are part of the abstract data type. It finds candidates by constructing a graph from an internal abstract syntax tree with the procedure and structure types being the nodes of the graph. The references by the procedures to the internal fields of the structures are the edges. The set of connected components in the graph form the set of candidates.

When recovering object instances, OBAD assumes they are implemented as a group of procedures (object behaviour) that reference a common set of global and other external variables (object state). Again, OBAD finds candidate objects by constructing a graph from the program's abstract syntax tree. For object recovery, the procedures and external variables are the nodes of the graph and the references by the procedure to the variables are the edges. The set of connected components in this graph form the set of candidate objects. In addition, OBAD can combine two types of recovery by retrieving connected components from the union of the two graph structures. Connected components that contain external variables or structures, but not both, are candidate object instances and abstract data types respectively.

3.3.2 Specification Driven Criteria

Canfora et al [Canfora94] propose a number of specification driven candidate criteria for isolating code fragments which implement functional abstractions in large programs. The function to be isolated is partially specified in terms of its set of input and output data using first order logic formulas called preconditions and postconditions. The preconditions express the constraints that must hold on the input data to allow the execution of the function. The postconditions express the condition that will hold on the input and output data after the function is executed. Other conditions, which bind the execution of a function in the context of a program, are also considered. The formal specification of a function is used with structural techniques based on the program dependence graph. In addition, human interaction is required to trace the data and conditions of the specification into the program variables and predicates.

Three types of conditioned software components are identified and extracted. These are:

- conditioned functions - given a program P and condition C, the conditional function F(C) consists of all the statements and predicates in P that are control dependent on a program predicate p implementing the condition C. A control dependence graph traversal algorithm is presented which searches for a predicate node p corresponding to the precondition or the binding condition of the required functional abstraction. The nodes that are control dependent on p are candidates to implement the functional abstraction and are isolated and executed. Human interaction is required to identify the program predicates implementing the condition.
- conditioned programs - Given a program P and a condition C, the conditional program P(C) is an executable program containing all the statements and predicates of P that can be executed when the condition C holds true. An algorithm traversing the control dependence graph is used to identify all the nodes corresponding to statements and predicates of P that can be executed whenever a given condition holds true. This method is useful for isolating different behaviours of a program module specified by a precondition composed of the disjunction of several conditions.
- conditioned slices - this is a generalisation of the quasi static slicing paradigm (see Section 2.1.3). A conditioned slicing criterion is of the form (sout, Vout, C), where sout is a program statement, Vout is a set of program variables and C is a condition. A conditioned slice of a program P on a conditioned slicing criterion (sout, Vout, C) consists of all the statements and predicates of P that might affect the values of the variables in Vout just before the statement sout is executed, when the condition C hold true. Conditioned slicing can be used to identify different behaviours of a functional abstraction implemented by a program slice. A two phase

algorithm is exploited which first computes the conditioned program P(C) and then the conditioned slice by using the program dependence graph of P(C).

3.3.3 Overview of Receiver Based Approaches

The stricter requirements on the grouping of routines with their receiver types return a finer object set than the type based approach. However, due to the approaches used, such as typing, the same problems which have been identified with the type based approach (Section 3.2) are also applicable to this approach.

3.4. Data Persistence Approaches

The data persistence approach groups on the basis of the data files when an application is executed.

3.4.1 The Gall and Klösch Approach

Gall and Klösch [Gall92] have defined a reuse engineering life-cycle. Their process of reuse engineering consists of eight steps. Their object identification process [Gall94, Gall95] starts with a design recovery step generating low-level design documents including structure charts and dataflow diagrams. The dataflow diagrams constitute the basis for object identification. A couple of entities are then selected and candidate objects are identified for this set. Two kinds of entities are considered. These are:

- data store entities (DSE)- i.e. information stored in a file (for instance, persistent data) that is essential for the program.
- non-data store entities (NDSE)- non-persistent data which is still essential for the program, such a values calculated during run-time and printed.

An algorithm for identifying NDSEs on a declarative basis examines each entity of the set of DSEs and tries to find some relationships to other user-defined data structures. Candidate NDSEs usually exhibit some kind of record structure defined by the user. The second strategy for identifying NDSEs is via functional relationships between DSEs and other user-defined data structures. For this, all those procedures and functions which use or manipulate entries of DSEs are identified. Their use and or manipulation in a specific procedure and or function defines a functional relationship so that the manipulated or used data structure is identified as a NDSE.

The above steps result in a set of DSE and NDSEs and candidate objects. To complete the object structure, it is necessary to identify the relationships among the entities. There are two kinds of relationships, which are:

- special relationships - these are *is-a* and *part-of* relationships, used for modelling the structure of a system.
- general relationships - these relationships incorporate the functional relationships between entities represented by procedures or functions of the procedural system.

Therefore, as a result of this process, the following elements are identified:

1. object candidates (based on the set of DSEs and NDSEs).
2. attributes (entities of the object defining data structures).
3. service candidates (based on the general relationships).
4. object structures (according to the special relationships identified).

The object identification process enables an assignment of source code elements to elements (objects) of the Reverse generated object-oriented Application Model (FooAM).

3.4.2 Overview of Data Persistence Approaches

The data persistence approach groups its objects based on the assumption that similar operations will write their data to the same persistent data store. The approach is also aided if different operations use different data stores. The basis of this assumption is that the code is well designed and that the separation of different data sets will have been performed to assist with the structuring of the code. Unfortunately, this is not always the case as legacy code is often poorly designed when compared with existing standards of newly designed code. Furthermore, even if a code module was initially well designed prolonged maintenance may mean that the quality of the design can no longer be assumed.

3.5. Existing Structure Approaches

An existing structure approach uses the existing structures and translates them directly into new objects.

3.5.1 CARE

Caldiera and Basili [Caldiera91] have proposed a process model for the extraction of reusable components. The CARE (Computer Aided Reuse Engineering) system has been designed to support the process model. The component identifier consists of the model editor, which helps in defining and modifying the reusability attribute model, and the component extractor which applies the model to the programs. The component qualifier consists of the specifier, the tester, and the packager.

The Caldiera and Basili process consists of two phases: the identification stage and the qualification stage. The first phase is automatic but the second phase requires human intervention. These phases are now described in more detail.

Identification

Within the identification phase, program units are automatically extracted and made to be independent compilation units. These independent units are measured in three steps according to observable properties which relate to their potential for reuse:

1. Definition of the reusability attribute model: a set of automatable measures that capture the characteristics of potentially reusable components is defined along with acceptable ranges of values for these metrics.
2. Extraction of components: modular units are extracted from existing software and completed so that they have all the external references needed to reuse them independently.
3. Application of the model: the current reusability attribute model is applied to the extracted, complete components. Components whose measures are within the models range of acceptability values become candidate reusable components to be analysed within the qualification stage.

Qualification

The extracted components are analysed in order to understand them and record their meaning. The components are packaged by associating with them a reuse specification, a significant set of test cases, a set of attributes based on a reuse classification scheme, and a set of procedures for reusing the component. This phase consists of the following steps:

1. Formal specification: a precise description of what the component does is generated and some assurance is obtained that the component meets the requirements.

2. Testing: Test cases are generated, extracted and associated with components. If it is likely that a component needs a wrapping to be executed, the testing step generates this wrapping.
3. Packaging: The extracted candidates are stored in the component repository along with their functional specifications and test cases. The component repository is actually a database of experience in which information on software products, processes and measures of aspects of them are stored.

3.5.2 ObjectOry

ObjectOry is a tool which supports object based forward development as well as reengineering. Jacobson [Jacobson91] uses the ObjectOry tool to show how an object oriented development method can be used to gradually reengineer an old system. The approach he uses for the identification of objects is as follows:

1. Identify the parts of the system that will be re-implemented using object oriented techniques - to achieve this a dependency graph is generated in the form of a directed graph ($D_{primitive}$, E_D,g(D_{pi},D_{pj})).

   Where $D_{primitive}$, is the set of all primitive descriptive elements,

   E_D is the set of arcs between the nodes.

   g is a function that associates an arc with an ordered pair of nodes. The function represents a dependency between the primitive description elements.
2. Prepare an analysis model of the part to be exchanged and its environment - in this case it is only necessary to concentrate understanding to a limited part of D.
3. Map each object to the old implementation of the system - two subsets are devised:
   i) D_X, which represents the part of the model that definitely will be implemented with the new technique.
   ii) D_{env}, analysis objects will serve as wrappers of the old system. They represent objects that D_X is related to.
4. Iterate the previous step until the interface between the part to be exchanged and the rest of the system is acceptable - this is achieved by examining the partition of $D_{primitive}$. Through changing the set D_X, it is possible to get a set D'_X that, through D'_{env}, gives a better interface between the old and the new environment.
5. In parallel:
   i) Design the new system and its interface to the rest of the old system.
   ii) Modify the old system and add an interface to the new subsystem.
6. Integrate and test the new subsystem and the modified old system.

This approach is different from many that have already been described in that the objects identified are re-implemented. Despite this difference, there are similarities within parts of the method, particularly within the identification steps. However, there are other distinctions resulting from the use of the forward engineering process, such as the need for additional testing.

This approach investigates potentials for minimising interfaces between the new components of the system. It does not, however, apply the concept assignment process within the method. Therefore, there is no assurance that the full functionality of a specific function of the original systems is encapsulated within the new object and still not partially provided by the original system.

3.5.3 The R^3A Approach

Yang [Yang97] supplements the object identification process using a tool called R^3A (Reverse-engineering Reuse Redevelopment Assistant). This tool uses transformation technology and a wide spectrum language (WSL). The following approach is proposed to enable the identification of reusable components through program understanding:

1. Translate a program in a source language into R^3WSL.
2. Use initial tidy-up transformations to 'clean up' the target program in R^3WSL in order to reduce the redundant statements introduced during translation.
3. Look for functionally self-contained modules. A code module, a function or procedure in the original code, is potentially a self contained module. A reusable component may well be obtained from one of the above modules. A module that is not a function or a procedure may also be transformed into an abstract data type, and hence is also a candidate reusable component. An abstract data type may be formed by looking for a closure of a group of variables and a group of procedures or functions.
4. Using each module obtained from the above process, program transformations are applied to reverse engineer the module into its high-level representation, in this case an entity relationship model.
5. The obtained entity relationship models together with the original code are used by the semantic analysis tool to generate semantic predicates and interface predicates for a reusable module in terms of its pre-conditions, post-conditions and obligations. These predicates are used to describe implicit semantics, characteristics, and interface requirements of each software component explicitly.
6. Store a reusable module in the reuse library maintaining a formal link between the reusability module and its high abstraction level representation.

3.5.4 Overview of Existing Structure Approaches

The use of an existing structure is one of the simplest approaches to selectively replacing key areas of the program. The simplicity of the approach stems from the fact that the interface to the new object will be similar to the one to which it will replace. The disadvantage of the approach is that there is little opportunity for undoing the problems associated with poor design and continual maintenance. In the data persistence approach described above (Section 3.4) the problems associated with legacy systems were described. One of the advantages of replacing modules is the opportunity to provide preventative maintenance. However, this approach does not include the opportunity to greatly restructure the existing code and, therefore, extensive preventative maintenance is not realised within this approach.

3.6. General Approaches

General approaches use a selection of some of the approaches described above as well as providing a set of guidelines as to how the objects should be grouped.

3.6.1 Sneed's Extraction of Object-oriented Specifications

Sneed [Sneed92] in the early 1990s defined an approach for object-oriented reengineering. This approach is called REORG. In the REORG reengineering process, there are 10 steps to transforming procedurally structured COBOL programs into object-oriented ones. These steps are as follows:

1. Static analysis of the source code
2. Inverse transformation into a specification repository
3. Identification of the object types
4. Analysis of the access paths
5. Analysis of the data usage
6. Relocation of instructions to objects
7. Analysis of the data flow
8. Marking of inherited attributes
9. Identification of the messages
10. Generation of an object-oriented program frame.

As a result of the application of this approach, Sneed set about refining this approach. This involved altering the process from a mainly bottom-up approach to mainly top-down. For this modified approach, Sneed[Sneed96] defines a five step process. These steps are:

1. Object selection
2. Operation extraction
3. Feature inheritance
4. Redundancy elimination
5. Syntax conversion

Sneed [Sneed95] believes that objects can be identified by assessing the different software components. As a consequence of object identification several types of objects are identified and stored together with the names and descriptions of their attributes. These are:

- user interface objects - derived from the panels
- information objects - derived from the database descriptions
- file objects - derived from the JCL
- record objects - derived from the programs
- view objects - derived from the programs
- work objects - derived from the program work area
- link objects - derived from the program parameters

The next step of the analysis process is to locate and extract the operations upon the objects. This is achieved by examining the cross references. These are then checked against the pre-defined objects. The following heuristics are applied:

- If an operand of an operation is an attribute of a given object, either directly or indirectly via a re-definition, then the operation is assigned to that object
- If any operand is set, then the operation is of the type UP_DATE
- If all operands are only used then the operation is of type RETRIEVAL
- If all operands are used and any are predicates in a condition, then the operation is of type QUERY.

Having extracted the operations performed on the object the following step is to depict their connections. Operations may also require data from other foreign objects; their parameters. Thus it is necessary to scan all operations belonging to a specific object type and to capture the foreign variables which are placed in the interface table. For each object from which an operation derives data, an interface is specified. The interface contains the name of the receiving operation, the name of the target objects, and the names and types of all variables obtained from the source object.

The final analysis step is to capture and document the sequence in which the operations are executed. In the Sneed model the information gathered acts as a basis for re-implementing the system.

3.6.2 RE^2 Reuse Reengineering Process

The key role of this paradigm is to define a framework where relevant methods and tools for partial solutions can be linked together. A reuse reengineering process called RE^2 has been defined by DIS (Department of Informatica e Sistemistica, at the University of Naples) and CSM (Centre for Software Maintenance, at the University of Durham). The RE^2 paradigm decomposes a reuse reengineering process into five sequential phases [Tortorel94, DeLucia95]. These are:

- Candidature
- Election
- Qualification
- Classification and Storage
- Storage and Display

The relationship between these phases is shown in figure 3.1.

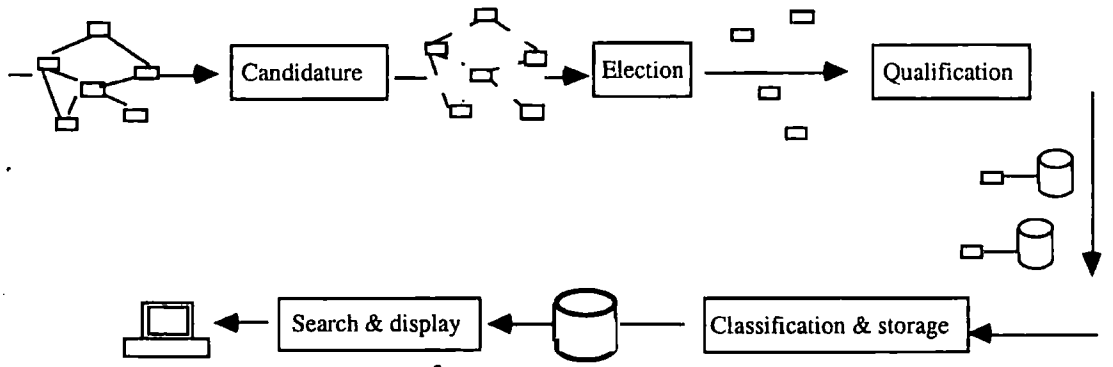


Figure 3.1: The sequential phases of RE^2

The first three phases are related to the identification, extraction and reengineering of software components for the production of reusable modules (and therefore of interest to this project). The latter two phases populate the repository and set up the environment for the retrieval and the reuse of modules during the development of its reference paradigm. Therefore these later phases are of only peripheral interest to this thesis. The first three phases are now described in more detail.

Candidature

The candidature phase produces a set of software components by using source code analysis techniques. Each set of software components are candidates for reusable modules when suitably de-coupled, re-engineered and possibly generalised. The candidature phase is organised into three steps:

1. Defining a candidature criterion to produce a first approximation of the set of reuse-candidate modules. This also entails the definition of the model of the system (typically a program representation) to apply the criterion and the information to make up an instance of the model.
2. Defining and setting up a reverse engineering process to extract a set of software components from code and make up an instance of the model defined in the previous step.
3. Applying the candidate criterion to the particular model instance to produce the set of reuse-candidate modules.

In RE^2, the software engineering principles of abstraction and information hiding are assumed as a guideline to define the reuse reengineering process. Consequently a candidate criterion should automatically produce a first approximation of the sets of components to extract from a software system. There are three search primitives from which candidature criteria can be defined: Isolation, Aggregation, and Generalisation.

Isolation - consists of breaking a large monolithic system component that implements more than one abstraction into a set of components each of which implements one abstraction.
Aggregation - consists of linking together several low-level system components (i.e. routines, functions, paragraphs, data types, variables and chunks of code) whenever these contribute to implement a higher level abstraction
Generalisation - consists of assigning a module with a higher generality, thus extending the range of problems it can solve. Generalising a module involves storing it in a 'generic' form from which several specific modules can be instantiated.

Election

The election candidate phase transforms reuse-candidate modules into reusable modules by de-coupling each set of software components from the external environment and clustering them into a module according to a pre-defined template. This phase consists of three steps:

1. Defining a module template (according to the primitives provided by the particular programming language considered and the concepts of abstraction and information hiding) for reengineering the reuse-candidate modules [Cimitile98].
2. Defining and setting up a reengineering process for de-coupling the components from the external environment.
3. Defining and setting up a reengineering process for clustering the components in the template.

Not all the meaningful candidate sets will be included in the reusable modules, some of them will be discarded during the election phase because of the complexity and the cost of the reengineering operation needed to de-couple and cluster them. Only reusable candidate modules that can be associated with human oriented concepts are de-coupled and reengineered.

Qualification

The qualification phase produces the interface and functional specifications of the reusable modules obtained in the election phase. The qualification phase is organised into three steps:

1. Defining or choosing a specification formalism expressing the semantics of a reusable module and how it should be used.
2. Defining and setting up of the functional reverse engineering process to produce the functional and the interface specifications from the source code module and to express them according to the defined formalism.
3. Testing and fixing the specifications produced to ensure their correctness and consistency with the code.

3.6.3 SRE Approach to Identifying and Extracting Components

At Andersen Consulting within the Strategic Technology Research Group a set of tools have been constructed for identifying and extracting reusable components from large legacy systems [Ning93]. The toolset, called SRE (System Renovation Environment), encapsulates the experience in the area of architecturing distributed tools developed for teams of people analysing a large amount of code.

The tools provide five distinct services. These are workspace management, system level analysis, data model recovery, concept recognition, and program level analysis. These are now described in further detail.

System level analysis is supported within SRE by the following types of system analysis approaches:

- System inventory - SRE helps the analysts to identify and link system source files into the tools knowledge base.
- System analysis - SRE analyses JCL scripts and CALL statements in the COBOL source code to recover data flow relationships between programs and data files and calling relationships among programs.

- System browsing - SRE uses a matrix browser to present a program-to-data-file in a tabular form. Moreover matrix cells can be selected to allow navigation to relevant program modules, files and JCL code.

The data model recovery process within SRE involves the identification of a virtual data model based on the analysis of data record mappings in the DATA DIVISION, data assignments in the PROCEDURE DIVISION, and data flows between programs through files. This virtual data model suggests how the data record declarations and names should be restructured / rationalised to make the system-level data view more consistent and less redundant.

Concept recognition is achieved through a knowledge based approach which has been developed to automate the recognition of functional patterns within the code. This work is based on Ning's PhD thesis [Ning89].

Program level analysis amounts to the automated understanding capacity available to assist program analysts. This includes:

- parsing, and program text browsing
- flow analysis
- complexity analysis and anomaly detection
- program segmentation

Finally, the design of the distributed execution architecture allows multiple application modules to run on different workstations along different execution threads and processes. The design is based on the software bus concept to facilitate process integration. This has a number of advantages including performance optimisation, openness, and support for multiple users.

The program segmentation process of program level analysis is achieved by five main operations. These are:

- Select statements - user selected statement objects
- PERFORMed statements - based on a PERFORM hierarchy
- Condition-based slice - the user specifies a logical expression that forms the basis of the slice
- Forward slice - given a variable and a slicing range, it retrieves statements in the range that could potentially be affected by the variable
- Backward slice - given a variable and a slicing range it returns statements in the range that can potentially affect the value of the variable.

More recently Ning [Ning96] has been investigating component based development models which concerns process, methods and tools that support software development from reusable components.

3.6.4 Overview of General Approaches

The advantage of a general approach can be realised in a number of ways. For instance, they may include a number of different approaches in recognition that not all approaches will work equally well with all types of code. Differences may result from the development approach selected or the implementation language used. However, the disadvantage of such an approach is that it includes a high degree of decision making on behalf of its user. Thus such an approach is more difficult to use than an approach that adopts a single object identification process. Therefore, it is important to strike the right balance between the flexibility of the approach and the skill level required of the user.

3.7. Summary

Within this chapter a number of different approaches to reengineering for reuse have been described. These approaches have been grouped into 6 different categories. At the end of each of the categories, some of the benefits and problems associated with the approaches were discussed.

In the following chapter a new approach will be proposed. This will be a method which is usable to maintainers who may not have a familiarity with existing reverse engineering practices. Many of the techniques described in this chapter are included within the method.

Chapter 4. The Method

The development of this method has resulted from taking a number of techniques from the RE^2 project and constructing them into a method using RECAST as a framework. The work described in this thesis contains a new method that modifies, extends and integrates the other projects. The work has been specifically targeted towards large COBOL applications. This work analyses the calling structure of the code (termed PERFORMs) and the interactions of its data items. An overview of the COBOL language will now be given with regard to these two features.

The COBOL language consists of two forms of proceduralisation statements, these are termed PARAGRAPHs and SECTIONs. A SECTION can consist of a number of PARAGRAPHs. Execution of PARAGRAPHs is based on sequential execution of the source code instructions. After the last statement of SECTION the execution will continue to the first statement of the proceeding SECTION. This type of continued execution is termed 'fall through', but this can be prevented through the use of an EXIT statement at the end of a SECTION. Changes in flow of control to COBOL PARAGRAPHs is achieved by a GO TO statement. GO TO statements can be used within or externally to the SECTION. The flow of control changes between SECTIONs through a PERFORM statement.

Data items within the COBOL language are hierarchically arranged. Those at the top of the hierarchy are termed GROUP ITEMs whereas those lower in the hierarchy are termed ELEMENTARY ITEMs. Changing a value of a GROUP item may also have an effect on the ELEMENTARY ITEMs. All data items defined within a program file within COBOL are global.

The new reuse reengineering method consists of 10 steps. The method identifies areas of potential reuse within legacy systems. These identified potential reuse areas are termed *candidate reuse units*. During the later stages of the method high level descriptions of their functionality are given to the candidates reuse units and decisions are made regarding their potential for reuse. Those identified as suitable are then termed *reuse candidates*. The detailed stages of the method are as follows:

STEP 1. Generate a PERFORM graph from the source code - the calling structure of the code is analysed and represented in the form of a graph.

STEP 2. Generate a dominance tree from the PERFORM graph - a tree of the PERFORM graph is formed based on the hierarchical nature of the procedural calls.

STEP 3. Identify candidate reuse units from the dominance tree - subtrees within the dominance tree are located to form candidate reuse units.

STEP 4. Identify data dependencies within the source code - data items used within the source code are categorised depending on whether they are read or written.

STEP 5. Identify data inter-relationships between subtrees - data items used across different subtrees (candidate reuse units) are highlighted.

STEP 6. Identify potential reuse candidates from users / designers of the code - designers are questioned as to their expectations of what reuse candidates are likely to be identified.

STEP 7. Identify potential simplification procedures to assist encapsulation - anomalies between located and expected reuse candidates are investigated.

STEP 8. Select subtree(s) to form reuse candidates using graph slicing - an initial list of reuse candidate is produced.

STEP 9. Identify data items in reuse candidates that would reduce data interactions - relocation of data items involved within interactions are considered.

STEP 10. Identify SECTIONs where slicing could assist separation - splitting SECTIONs is considered to reduce interactions between reuse candidates.

Each step of the method forms a subsection within this chapter. Each subsection follows the same format. At the beginning of each is the step's objective. This is followed by a description of how the objective may be achieved. Small examples are used throughout to demonstrate the approaches used.

The steps are now described in detail.

4.1. Step 1. Generate a PERFORM Graph from the Source Code

4.1.1 Step Objective

- The objective of Step 1 is to identify the calling structure of the code on a SECTION basis.

4.1.2 Approach

A PERFORM graph is a *call-directed-graph*. A call-directed-graph of a code module is formally defined as a directed graph CDG = (N,E) where N = S $\cup$ PP is the union of S set of the entry SECTIONs and the set PP of all SECTIONs, and E is the PERFORM Relation ({s} $\cup$ PP) x PP.

The step consists of two tasks. These are:

a) check preconditions for producing the PERFORM graph
b) generate the PERFORM graph

A description of each of these tasks will now be given.

a) Check preconditions for producing the PERFORM graph

The generation of a true PERFORM structure of COBOL relies on the following conditions holding:

- GO TOs are not used between SECTIONs
- GO BACKs are not used between SECTIONs
- each SECTION must have an EXIT statement at the end of the SECTION to prevent 'fall through' to the following SECTION

If any of these conditions do not hold then the code should be restructured before the PERFORM graph is generated.

b) Generate the PERFORM graph

A PERFORM graph is constructed by analysing the PERFORMs for each SECTION. For each PERFORM in a SECTION, a link is made between the calling and called SECTIONs. These caller / callee relationships are referred to as the PERFORM graph pairs. Where more than one PERFORM is issued to a particular SECTION within a SECTION, then the duplicate PERFORM is recorded numerically (i.e. caller, callee, number_of_times_performed). The statements PERFORM n TIMES are not considered as these inline PERFORMs are more like loop statements.

The following code sample (only SECTION name and PERFORM statements are visible) would yield the PERFORM graph shown to the right of Figure 4.1.

```
:
PROCEDURE DIVISION
:
-------------------------------
A000 SECTION.
:
  PERFORM B000
  PERFORM C000
 EXIT.
-------------------------------
B000 SECTION.
:
  PERFORM B100
  PERFORM B200
EXIT.
-------------------------------
B100 SECTION.
:
EXIT.
-------------------------------
B200 SECTION.
:
EXIT.
-------------------------------
C000 SECTION.
:
EXIT.
-------------------------------
```

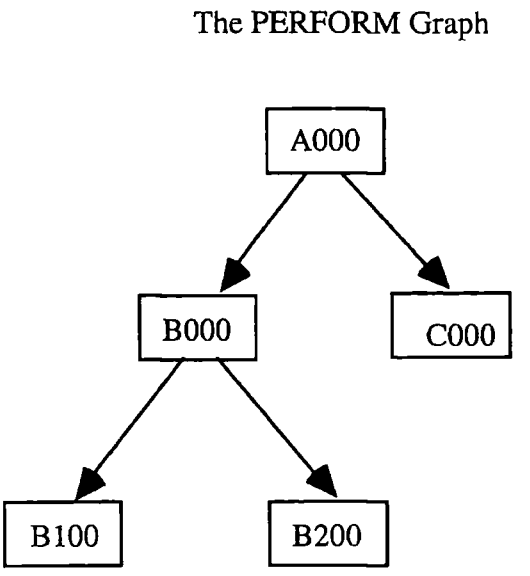


Figure 4.1: Generating a PERFORM graph from the source code

As shown in figure 4.1 the PERFORM graphs can be represented graphically. The SECTIONs are represented as nodes and the PERFORMs as edges. The graph provides a view of the complexity of the program in terms of the calling structure. In addition, problem nodes can be identified such as those which are PERFORMed by many SECTIONs. The observations made from this graph can then be used to assist a number of the later stages of the method.

4.2. Step 2. Generate a Dominance Tree from the PERFORM Graph

4.2.1 Step Objective

- The objective of this step is to identify the dominance relationships between SECTIONs.

4.2.2 Approach

A dominance tree is created from a call-directed-acyclic-graph. Two relationships can be identified between the nodes of the graph. These are strong and direct dominance [Hecht77].

This step is composed of two tasks. These are:

a) Remove cycles from the PERFORM graph
b) Generate the dominance relations

These are now described.

a) Remove cycles from the PERFORM graph

As indicated above, if a PERFORM graph is to be analysed accurately then its cycles should be removed before dominance tree analysis is initiated. Thereby a call-directed-acyclic-graph (CDAG) is obtained from a call-directed-graph (CDG). Cycles can occur within the source code between two or more nodes and are a product of carrying out static rather than dynamic analysis. Cycles can be removed by collapsing every strongly connected subgraph (those nodes contained within a cycle) into one node. When an individual node is contained within two cycles then both sets of nodes should be collapsed.

b) Generate the dominance relations

The dominance relations are obtained in the following way. In a CDAG, a node *px* dominates a node *py* if, and only if, every path from the initial node *s* of the graph to *py* includes *px*. In a CDAG, a node *px* directly dominates a node *py* if, and only if, all the nodes that dominate *py* dominate *px*. 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 it is the only node that calls *py*.

The PERFORM Graph (extended from step 1)

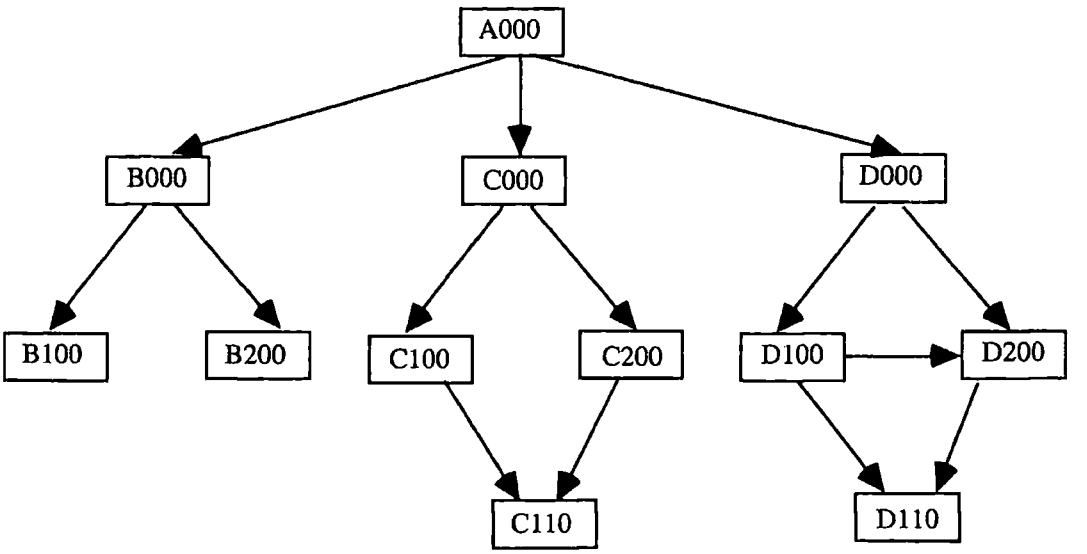


Figure 4.2: A PERFORM Graph

Figure 4.2 shows a simple PERFORM graph that has been extended from the example code given within Step 1. Using the techniques described above, this PERFORM graph is converted into a dominance tree.

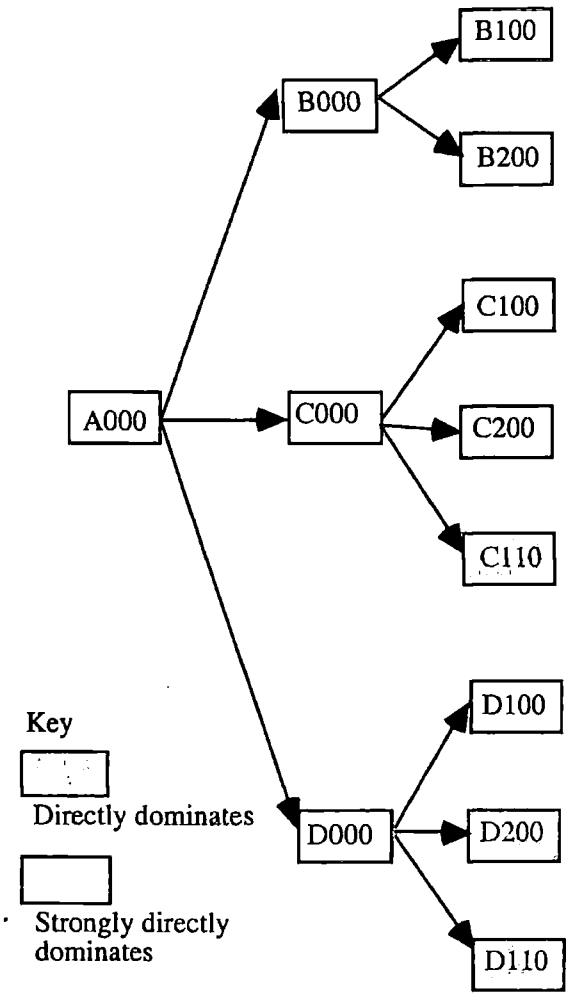


Figure 4.3: A Dominance Tree

As shown in Figure 4.3, the dominance tree can be represented graphically. This provides a visual indication of the areas of potential reuse within the program.

4.3. Step 3. Identify Candidate Reuse Units from the Dominance Tree

4.3.1 Step Objective

- The objective of this step is to identify as many candidate reuse units as possible. In addition, units at different levels of granularity should be considered.

4.3.2 Approach

This Step consists of a single task:

a) identify candidate reuse units at varying levels of granularity

A description of this task is now given.

a) Identify reuse candidate units at varying levels of granularity

The areas of potential candidate reuse units are those subtrees that have a number of strongly dominated nodes. This relationship represents a unique path through a set of nodes. The inclusion of directly dominant nodes will mean that PERFORMs externally of the candidate reuse unit will exist. Thus the strong dominance relationships can be viewed as representing sub-components of the dominating node. Visual analysis provides a simple way of identifying the candidate reuse units. However, automated analysis of graphs is also possible.

From the example dominance tree shown in Figure 4.3, three candidate reuse units (subtrees) can be identified. These are highlighted in Figure 4.4. Those candidate reuse units which contain the fewest directly dominant nodes (the shaded nodes) will be the easiest to reuse (with relation to the PERFORM structure) as the PERFORM statements are restricted to within the individual candidate reuse unit.

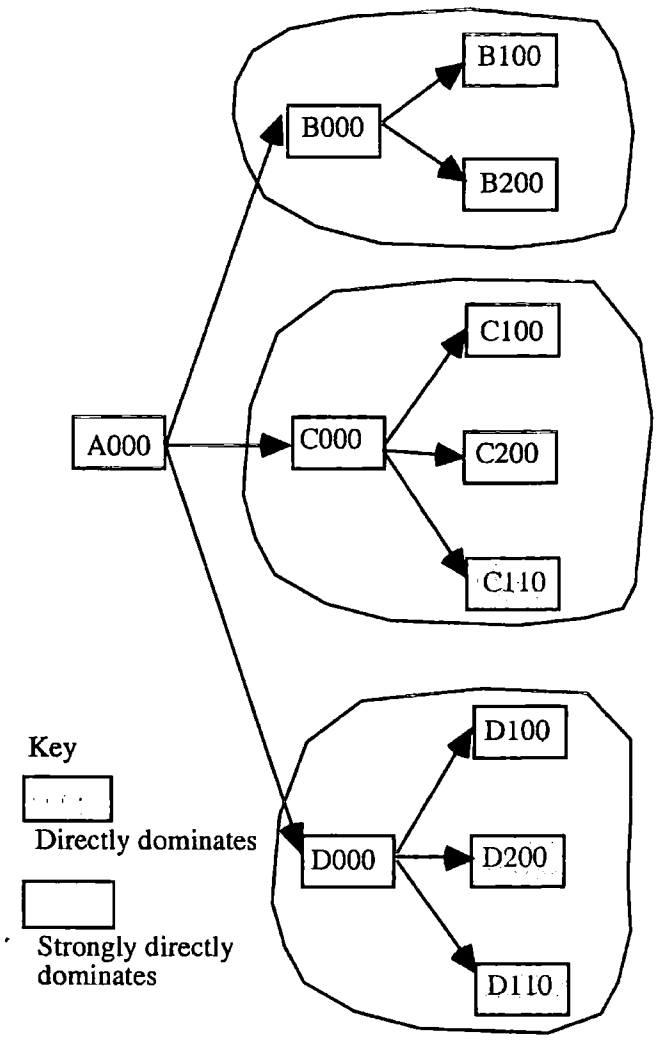


Figure 4.4: Reuse Candidate Units (subtrees)

Within larger reusable subtrees, there may be smaller subtrees that can also be potential candidate reuse units. The decision as to which candidate reuse units to extract, and at what level of granularity are issues that will be discussed later in this chapter.

The result of carrying out this step will be a number of potential candidate reuse units. It is advisable at this stage to identify a number of levels of granularity and to wait to investigate the effect of the data dependencies (Steps 7 - 10) before deciding with which candidate reuse units to proceed.

4.4. Step 4. Identify Data Dependencies within the Source Code.

4.4.1 Step Objective

- The objective of this step is to identify which data items are accessed within each SECTION of the source code.

4.4.2 Approach

This Step consists of three tasks. These are:

a) Identify which data items are referenced within the code
b) Identify sets of GROUP items
c) Express the relationships between the members of the GROUP items

These tasks are now described.

a) Identify which data items are referenced within the code

All data items of the source code should be identified. For each SECTION the following information should be collected:

- which data items are referenced
- whether a data item is Created, Updated, Read or Deleted

For instance, an update operation in the COBOL language on the data item x would be as follows:

```
SET x TO 4.
```

Likewise, a read operation would be as follows:

```
IF x IS NOT = 3
```

b) Identify sets of GROUP items

By analysing the GROUP definitions in the WORKING STORAGE SECTION, the hierarchical nature of the data items can be investigated. For instance, consider the following COBOL construct:

```
01 INPUT-DATA.
    05 INPUT-DATE                          PIC 9(8).
    05 INPUT-DATE-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).
```

The code sample above shows how GROUP and ELEMENTARY data items are expressed within the COBOL language. The names of data items within SECTIONs do not take account of the relationships

of data that exist. The relationships between data items should be identified. The GROUP relationships are not explicit when matching SECTION's use of the data items by their associated name. In COBOL, GROUP items are indicated by having a level number of a higher value than its ELEMENTARY items where the ELEMENTARY item is a leaf node. For instance, 01 INPUT-DATA is a GROUP item with the ELEMENTARY item 10 INPUT-DAY. When identifying data dependencies these relationships are important. In the above example, for instance, updating INPUT-DAY will also have an update effect on INPUT-DATE and INPUT-DATA.

c) Express the relationships between the members of the GROUP items

By analysing the data item dependencies, groupings can be represented so that the relationships between data items are evident. Such relationship can be represented in a graphical format for manual visual identification but can also be identified automatically. For graphical representations these relationships are represented using the *consists*, *redefines* and *overlapping value* set relationships.

The consists relationship -	This relationship expresses the components (i.e. ELEMENTARY items) of a specific GROUP item
The redefines relationship -	This relationship expresses the sharing of the same logical memory space
Overlapping value sets -	This relationship expresses the use of data items to categorise a subset of the ELEMENTARY data item's values.

In Figure 4.5 the hierarchical relationship of GROUP to ELEMENTARY item is represented as a consists relation. This hierarchical relationship allows a form of data typing to be expressed.

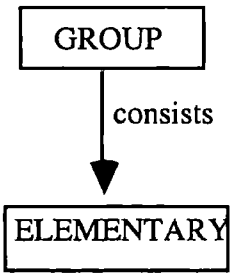


Figure 4.5: The hierarchical consists relation

From the same date example used within the task above it is possible to derive a number of *consists-of* data relationships. These are shown in Figure 4.6.

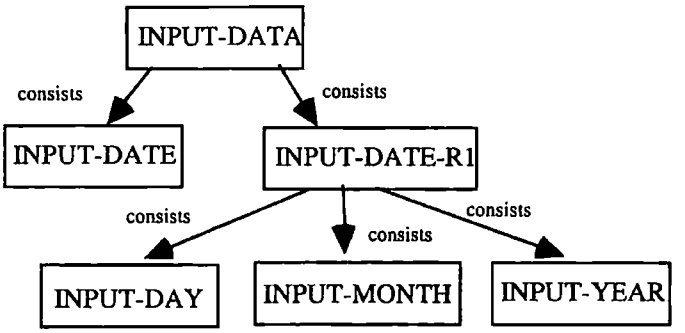


Figure 4.6: An example of the consists relationship from COBOL data typing

In Figure 4.6 it can be seen 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 candidate reuse unit's code. The absence or presence of part of the type hierarchy from the candidate reuse unit 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.

A further activity within this step involves the identification of overlapping data values. Overlapping value sets are additional grouping relationships that can be placed on the ELEMENTARY items by using their value settings. To highlight these issues a further example is introduced, that of months of the year. In this example a level 20 ELEMENTARY item of INPUT-MONTH for each month is defined i.e.

```
10 INPUT-MONTH.
        20 JANUARY          PIC 9(2).
        20 FEBRUARY         PIC 9(2).
        :
```

But additionally, the COBOL lists 20 WINTER_MONTH that groups a partial set of the other ELEMENTARY items. Using the example of days of the week from above overlapping values are evident where use is made of a style of enumerated typing. For instance:

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

In the above example the last three data types overlap. It is possible to represent these overlapping values graphically as an extension of the consist graph (Figure 4.7). In this example, the weekend data type groups its two overlapping values. The notation can be further extended, when the candidate reuse unit does not comprise all the possible values, by adding colour coding.

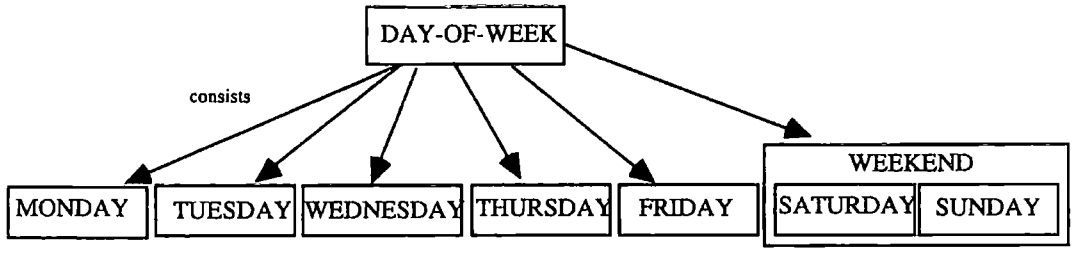


Figure 4.7: An example of overlapping values within the consists graph

In some cases the shaded data items (Figure 4.8) are used outside the candidate reuse unit.

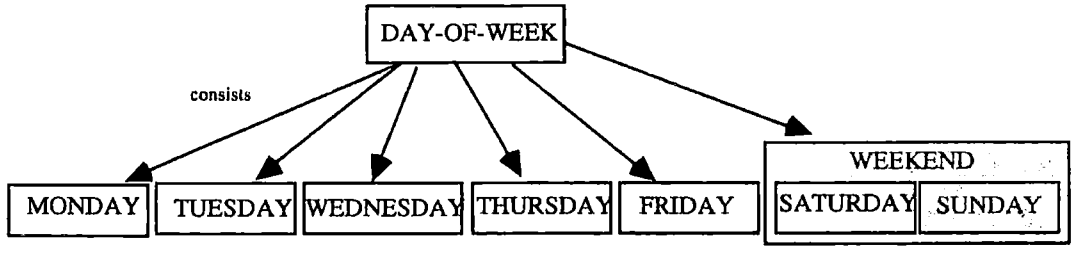


Figure 4.8: Values used outside of the reuse candidate

Therefore, this provides the justification of why it is important to record both the Weekend and Sunday and also Day-of-week data items, as each is potentially affected by the remainder of the code. This indicates a further area where the simplification of the reuse candidate unit's interface could be achieved by reengineering.

Further data interrelations also exist through the redefines relationship. An example of the usage of such a construct can be found in the above typing sample. i.e.

```
05 INPUT-DATE-R1 REDEFINES INPUT-DATE
```

The effect this has on the consists graph (Figure 4.6) is shown in Figure 4.9. In addition, in some cases, it is important to consider the reverse relationship 'is-redefined-by'. In the example below, this relationship is Input-date *is-redefined-by* input-date-R1.

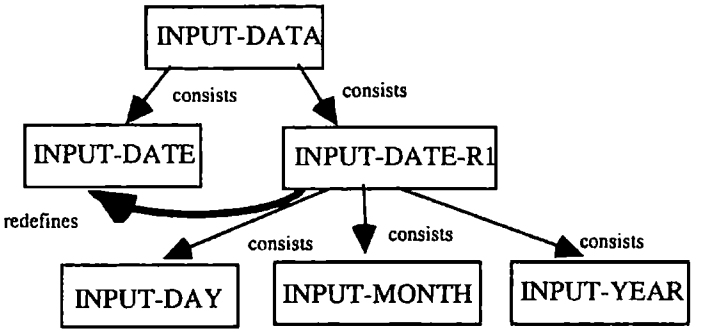


Figure 4.9: An example of the redefines relationship

The result of carrying out this step will be a complete analysis of the data usage within each of the SECTIONs of the source code. This includes the formation of the set of graphs identifying the data items redefines, consists and overlapping value set relationships.

4.5. Step 5. Identify Data Inter-relationships Between Subtrees

4.5.1 Step Objective

- The objective of this step is to identify subtrees which use independent data items.

4.5.2 Approach

This Step consists of two tasks. These are:

a) Identify areas of data independence
b) Use cluster analysis to identify the simplest interfaces between candidate reuse units

These tasks will now be described.

a) Identify areas of data independence

The SECTIONs that each data item is referenced by should be compared to identify groups of related data in order to identify any data independent parts of the program. In addition, the interactions between data items can also be investigated for the subtrees identified through dominance tree analysis. The candidate reuse unit can either be subgraphs identified through the analysis of the dominance tree or those identified as data independent in this step.

b) Use cluster analysis to identify the simplest interfaces between components

Grouping of data items is carried out with the use of cluster analysis [Everitt93]. The data interactions are measured using some of the simplest similarity coefficients for dichotomous variables, where each variable has only two values. In this case the values are either present or absent (represented as 1 and 0). In particular, two approaches are applied. These are the matching coefficient and Jaccard's coefficient. The matching coefficient is the ratio of the total number of variables on which the two individuals match, to the total number of variables. The Jaccard coefficient is calculated using the matching coefficient but 'negative' matches are ignored. Negative matches in this instance represent a match based on the absence on a specific data item. The absence of data items, in this case is less important and therefore can be ignored in favour of positive identifications. Using this approach a matrix is derived and differences between each SECTION data usage are compared.

The clustering approach attempts to define candidate reuse units objects that offer a single functionality that is large enough to make its reuse worthwhile. Candidate reuse units that are too small may cost more to reuse than those developed from scratch. On the other hand, a candidate reuse unit which is too large (possibly offering more than one functionality) may be difficult to understand and thus have limited reuse potential.

The basis of a clustering technique for use as an approach to re-modularise code assumes that specific functionality within a program can be identified by the data sets manipulated. The approach assumes that data items can be grouped into disjoint (or mostly disjoint) sets by investigating the interactions of the data items. Data interactions are the interface between re-modularised candidate reuse units, where two or more candidate reuse units access a specific data item. This is particularly problematic in COBOL where all data is global. Thus, by forming candidate reuse units based on their 'similarity' i.e. grouping data items that frequently interact, clusters can be formed to restructure source code. These new candidate reuse units aim to make the legacy code more maintainable in the future and therefore be supportive to the evolution process.

To gain an understanding of the results of the techniques of data clustering, the following approaches are evaluated. These are:

1. An assisted approach - This approach uses candidate reuse units already partly defined, for instance, based on the dominance tree structure. In this case, the broad outlines of candidate reuse units gained from the subtrees of the dominance tree are used as initial solutions. The results of the cluster analysis in this case will be based on the original dominance tree subtrees, but will indicate possible cases where data items should be relocated to SECTIONs excluded from a specific candidate reuse unit. This approach is much faster than the non-assisted one since the process starts with partial solutions. It is

also useful for evaluating the suitability of a specific level of granularity to choose for a reuse candidate unit.

2. A non-assisted approach - This composes candidate reuse units from scratch, for instance, matching data usage. The process is time consuming as in this case no partial solution is given. In addition, it is feasible to have many similarly ranking results. For instance, a number of possible data item clusters that offer different candidate reuse unit formations, but the same number of interactions between their interfaces. For this reason, with this approach, it is often necessary to later use further evaluation criteria to assess the reuse potential.

The suitability of the resulting clusters is assessed by the number of data items shared between the proposed candidate reuse units. Assuming that the reuse reengineering method has identified 2 candidates, C1 and C2, and that C1 uses the following data items {INPUT-YEAR, INPUT-MONTH, INPUT-DAY, TMP_C1} and C2 uses {INPUT-YEAR, INPUT-MONTH, INPUT-DAY, TMP_C2}, then using the above approach table 4.1 would be constructed:

Data Item	Use within C1	Use within C2
TMP-C1	1	0
TMP-C2	0	1
INPUT-YEAR	1	1
INPUT-MONTH	1	1
INPUT-DAY	1	1

Table 4.1 Data items used within the candidate reuse units

This process is initially carried out for the individual SECTIONs and then for the candidate reuse units. However, for brevity, only candidate comparisons are shown. The results of the analysis process can then be represented graphically. The graphical representations (Figure 4.10) are particularly useful in showing the interface between the candidate reuse units and those data items that are accessed only internally by specific candidate reuse units. Thus, these data items can be considered as local.

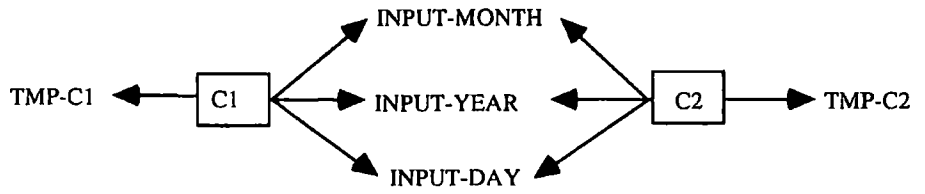


Figure 4.10: The interface between C1 and C2

Using the data usage information (from Step 4), it is possible to identify data items which are separated unnecessarily, for instance, which data items are updated (U) in different parts of the program to where they are read (R). These anomalies identified can be rectified later by restructuring (Step 9).

This step aims to identify the data usage of each of the data items. These are categorised into Update (U) and Read (R). When data is passed between candidates it is a two way process. When interactions of two reuse candidates are examined both candidates have the potential to read and update the data item. For example candidate 1 may read and update a specific data item, whereas candidate 2 may only read the data item. This would be represented as (RU,R).

To investigate the data interactions, nine relations are examined: ((RU,RU), (R,R), (U,U), (U,R), (R,U), (U,RU), (RU,U), (RU,R), (R,RU)). Of this set of relations, some are the same, they just refer to the specified order of the candidate reuse units. Thus, those relations that specify candidate ordering (and are therefore the same) are ((U,R), (R,U)), ((U,RU), (RU,U)) and ((R,RU), (RU,R)). Each of the relations refer, if they were to be redeveloped to a fully parameterised interface, to the input and output of the data flow between two candidate reuse units.

Assuming that the two candidates contained the following subsets of code:

Code within Candidate C1

```
IF INPUT_DAY = '31'
AND INPUT_MONTH = 'DECEMBER'
        ADD '1' TO INPUT_YEAR
        MOVE 'JANUARY' TO
INPUT_MONTH
        MOVE '1' TO INPUT_DAY
```

Code within Candidate C2

```
IF INPUT_YEAR = '1998'
        WRITE INPUT_DAY.
        WRITE INPUT_MONTH.
```

The data interfaces between the candidates would therefore be:

INPUT_YEAR	(U,R)
INPUT_MONTH	(UR,R)
INPUT_DAY	(UR,R)

Table 6.4 shows an example where the interface between two candidate reuse units (C1 and C2) is identified. The Xs represent cases for each of the nine relations where the candidate will require to input and/or output the value of a specific data item.

Usage	Input (C1)	Output (C1)	Input (C2)	Output (C2)
(RU,RU)	X	X	X	X
(R,R)	X		X	
(U,U)		X		X
(U,R)		X	X	
(R,U)	X			X
(U,RU)		X	X	X
(RU,U)	X	X		X
(RU,R)	X	X	X	
(R,RU)	X		X	X

Table 4.2: Usage translation to parameterisation of the interface

The data interfaces are then available for use within the later stages of the method where later reductions in the data interfaces between candidates are explored. The result of this step is an indication of data independent parts of the program and / or a list of data items that link two subtrees.

4.6. Step 6. Identify Potential Reuse Candidates from Users / Designers of the Code

4.6.1 Step Objective

- The objective of Step 6 is to apply a top-down approach to the identification of reuse candidates with the assistance of the users and designers of the code.

4.6.2 Approach

Steps 1-5 have so far analysed the low-level source instructions and attempted to represent subsets of high-level functionality. This approach is referred to as a bottom up approach. Within Step 6 a top down approach is employed.

This step is composed of two tasks. These are:

a) Use experts to describe the functionality of the candidates proposed.
b) Encourage experts to discuss improvements to candidates identified.

These tasks are now described.

a) Use experts to describe the functionality of the candidates proposed

The objective is to assist with the bottom-up approach for the identification of candidates reuse units (Steps 1 to 5) by considering the functionality of the system and therefore trying to gain an understanding of potentially which candidate reuse units could be present. This process is termed concept assignment as a description of the functionality is established for each of the reuse candidates units previously identified. Thus, techniques are employed, other than source code analysis, to identify potential reusable functionality within the system. The reusable functions identified by this approach are termed *reuse candidates* because they have a concept assignment (given by the experts) and may also have been identified as useful for future developments. The approach taken for this step comes from the RECAST method.

The two stages of the RECAST method used are:

- business users' views - by consulting designers and maintainers of the system (RECAST Step BUV-1)
- menus and dialogues - by executing the program to see it in operation (RECAST Step MD-1)

The results of top-down and bottom up analysis are then compared and matches are made between the reuse candidate units gained from the different approaches. Early comparisons can then be made toward considering the match / mismatch of candidates identified through the analysis of the code and those identified by the experts. Where mismatches are identified, further analysis should be carried out until the differences can be resolved.

One of the problems with involving the users of a system is that inaccurate or inconsistent information regarding the functioning of the system is often obtained. Since highly technical information is required the approach taken was to use the same user interviewing process described in RECAST, but to interview technical staff such as the maintainers and designers.

Their assistance is expected in two main ways. Through the interview process, they are encouraged to identify the elementary graph simplification processes (to be used in Step 7) and also their views are solicited as to what they believe are the reusable functions of the system. An example of the top down process reuse candidate identification process is shown in Figure 4.11. In this case, expert mappings show the relationships between specific SECTIONs which are identified by experts to be an 'Account Overdue Check'. These mappings are then used to derive a reuse candidate. Unless these expert mappings are investigated, there may not be an obvious relationship between the SECTIONs within the source code.

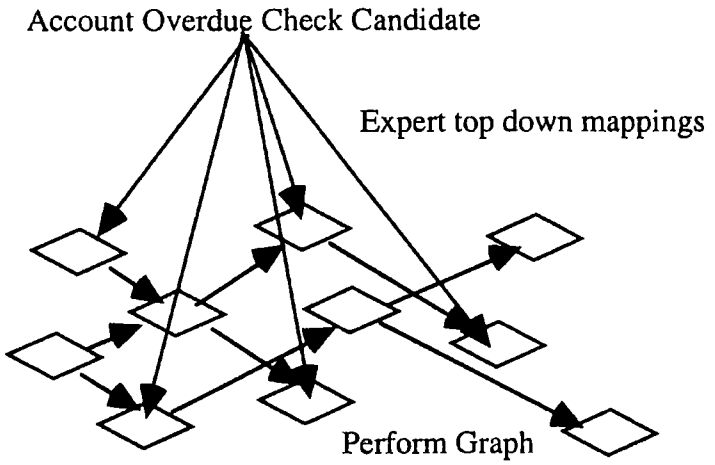


Figure 4.11: Expert Reuse Candidate Mappings

b) Encourage experts to discuss improvements to candidates identified

This process involves the identification of anomalies within the candidate reuse unit that are a result of the prolonged maintenance process. In this task experts are asked to evaluate if SECTIONs or data items appear to be misplaced. Any items identified are the cases for restructuring and dealt with during the later stages of the method.

The likely result of this step is the identification of a set of candidates of a far higher level of granularity that has so far been identified by the bottom up process. This will therefore be represented as a mismatch between what the bottom up process identifies as potentially reusable and those that the system designers have pin-pointed. In many cases the mismatches will represent only differing levels of granularity of reuse candidates. Thus, the two approaches will have identified two sets of results, which are consistent. The top down approach is useful to indicate what higher level representation should be identified. Thus, using this top down approach of predicting what is expected to be found, assists in grouping the lower level units from the bottom up approach.

4.7. Step 7. Identify Potential Simplification Procedures to Assist Encapsulation

4.7.1 Step Objective

- The objective of this step is to investigate heuristics for the simplification of the PERFORM structure and data interactions of the graph to reduce the complexity of its interactions and therefore assist the encapsulation process.

4.7.2 Approach

This step consists of two sets of tasks. Those tasks that are concerned with SECTION relations and those tasks that are concerned with data relations. These two sets of tasks as are as follows:

SECTION Relations

a) Locate non-functional requirements from a list of ones typical within the domain
b) Compose an application specific set of non-functional requirements
c) Identify actual SECTIONs within the code that implement the non-functional requirements
d) Analyse areas of high fan-in within code
e) Remove identified SECTIONs from the code

Data Relations

a) Identify data items not implementing specific functionality
b) Identify data items that are not updated throughout the code
c) Identify data items already external to the program

The details of each of these tasks follow.

It would appear that certain aspects of the code tend to hold the otherwise separate reuse candidates together. These aspects tend to represent the implementation of non-functional requirements. Due to their nature, non-functional requirements relate to a number of the system's functional requirements. Thus, non-functional requirements are often identifiable within the code as those forming areas of proportionally high fan-in.

In the COBOL language such SECTIONs with high fan-in are those which are PERFORMed by a large number of other SECTIONs. If the non-functional requirements can be temporarily excluded from the analysis process, the identification of areas of functionality within the source code is simpler. The reuse reengineering method therefore includes a number of simplification procedures to reduce the effect of these non-functional requirements.

Two forms of simplification procedures can be applied. These are the simplification of:

- SECTION relationships i.e. by removing those that are only concerned with non-functional requirements
- data relationships i.e. by removing those that are used for the implementation of non-functional requirements

These two approaches are separated into subsections below.

SECTION Relationships

In order to identify those data items or SECTIONs that can feasibly be removed without affecting functionality (Figure 4.12), parts of the RECAST method are used.

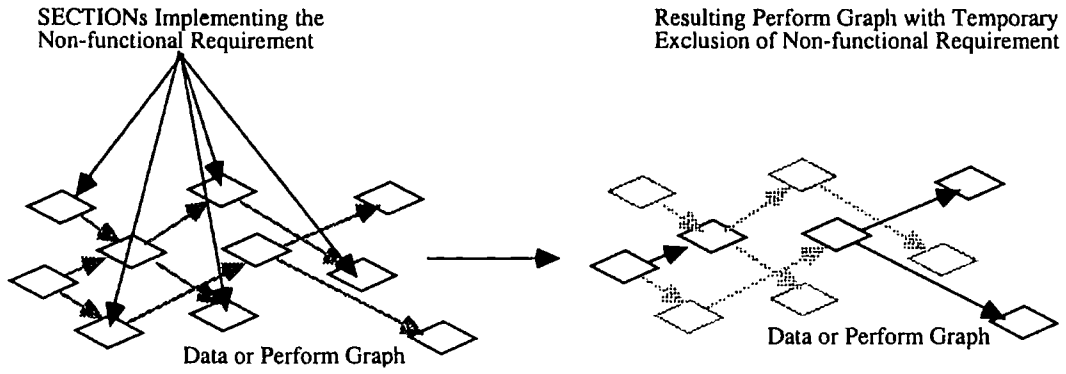


Figure 4.12: Example of a Temporary Removal of Non-functional Requirements

Two of the four stages of the RECAST method are used. These stages are:

- the logical data model - for the identification of the functionality and those data items or SECTIONs which can be removed as they do not describe the functionality of the program
- system processing entity life histories - for the identification of data items which are not updated in the program (therefore do not affect the way the code functions) so they can be removed from the analysis process

Activities from the RECAST method have been adapted and extended to specialise them for the reuse reengineering method's graph simplification requirements. A number of the resulting techniques are employed to locate the non-functional aspects of the code. These techniques can be thought of as individual tasks to complete within this step. These tasks for the simplification of SECTION relationships are as follows:

a) Locate non-functional requirements from those typical within the domain

A list of typical non-functional requirements is used to provide an initial indication of the type of requirements that could potentially be located within the code.

b) Compose an application specific set of non-functional requirements

The list is utilised by the users and designers of the code to draw up a new list of application specific non-functional requirements. If the original documentation is available, such as design and requirement

specification documents, these are used to assist with the generation of the application specific requirement list. Documentation is searched using a technique proposed for analysing the traceability links within documentation [Boldyref96].

c) Identify actual SECTIONs within the code that implement the non-functional requirements

The code is analysed to find the SECTIONs (or part of SECTIONs) that have been identified by the users and designers or those SECTIONs which have been located within the requirements documentation.

d) Analyse area of high fan-in within code

The PERFORM structure of the code is used to investigate those SECTIONs which have a high proportion of fan-ins (within the PERFORM graph structure these are the nodes which are PERFORMed many times).

e) Remove identified SECTIONs from the code

The SECTIONs which implement the non-functional requirements are temporarily excluded from the analysis process and the identification of reuse candidates is repeated to see if larger objects are obtained.

The SECTIONs and data items initially removed will be retained so not to misguide the re-implementation stage. Removal is during the analysis stages so that they do not obscure the more important SECTIONs and data items (describing the functionality of the program). It is important to reconsider the non-functional aspects at a later date so that an accurate assessment of the costs involved the final restructuring process can be gained.

In particular, targeting those data items and SECTIONs that individually have high connectivity to other SECTIONs is beneficial. However, it is not feasible to simply remove those SECTIONs and / or data items that are 'hot spots' of connectivity. When SECTIONs are to be temporarily removed from the analysis process there must be a sound justification for doing so. Those SECTIONs not representing the true functionality of the program are an example of SECTIONs that it is justifiable to remove. However, it is only appropriate to consider their removal if as well as being non-functional they also detract from the identification of functionality.

Data Relationships

A further approach that can be adopted to assist with graph simplification is the identification of data items that can be temporarily removed from the analysis process. A number of different approaches are

used for the selection of suitable data items. Specifically, the process involves the two tasks associated with data relations. These are the identification of data items not implementing specific functionality and of data items that are not updated throughout the code.

a) Identify data items not implementing specific functionality

Those data items that are already part of the interface of the original code module (as opposed to the set of reuse candidates) can be temporarily excluded from consideration without affecting the operation of the program. This approach is justified as these data items will not require reengineering as they are already part of the program's interface.

b) Identify data items that are not updated throughout the code

The data items that are never updated in the life-time of the application's operation have already been identified within Step 3. These data items being those which are only ever read. This approach can be justified as these data items can be considered as constant throughout the program. The removal of these two types of data item (Tasks a and b) further reduces the amount of data which aids the analysis of the interface between each of the reuse candidates.

c) Identify data items already external to the program

The removal of the data items which are already external to the program also aid the simplification of the data interactions. In this case, these data items cannot be local since they are used externally to the program. These can therefore be removed from any investigations aimed at identifying local data items.

The expected result of this step is a simpler graph, but which still represents the functionality of the source code with the exception of its non-functional aspects. The reduction of the complexity by hiding the non-functional aspects will aid the identification of candidate reuse units as the resulting graph is much easier to understand.

4.8. Step 8. Select Subtree(s) to Form Reuse Candidates using Graph Slicing

4.8.1 Step Objective

- The objective of this step is to select candidates which show they have potential for future reuse.

4.8.2 Approach

This step collates the information gained from the previous steps of the method regarding the potential for reuse of each of the candidates identified. During this step the most promising candidates are selected and their exact composition is found using slicing.

This step comprises two tasks. These are:

a) Collate all existing information on the suitability of all candidates. Make decisions as to which are the most viable.
b) Show exact composition of each candidate using graph slicing.

These tasks are now described below.

a) Collate information on the suitability of candidates and assess their viability

The criteria for the selection of reuse candidates is as follows:

1. The proportion of direct and strong dominance nodes. The more direct dominant nodes present within a candidate, the greater the cost of making the candidate reusable.
2. The complexity of the interface of the candidate. For instance, the number of data items which are needed to be made accessible externally of the candidate.
3. The 'usefulness' of the functionality to the company. For instance, if, or how often, the management foresee the reuse candidate being used within future developments.

b) Show exact composition of each candidate using graph slicing.

During this task the most promising candidate's exact composition is found using slicing. Specifically, subsets of the program are then obtained through slicing at the SECTION level. Traditional slicing involves the process of stripping a program of statements without influencing the result of a given variable. The same approach is used but slicing is carried out at the SECTION level. Thus, slices are based on sequences of PERFORMs between SECTIONs rather than for individual statements. In order to slice reusable subtrees, the top node of the tree (the root SECTION of the subtree) is located and all its children and grandchildren are identified.

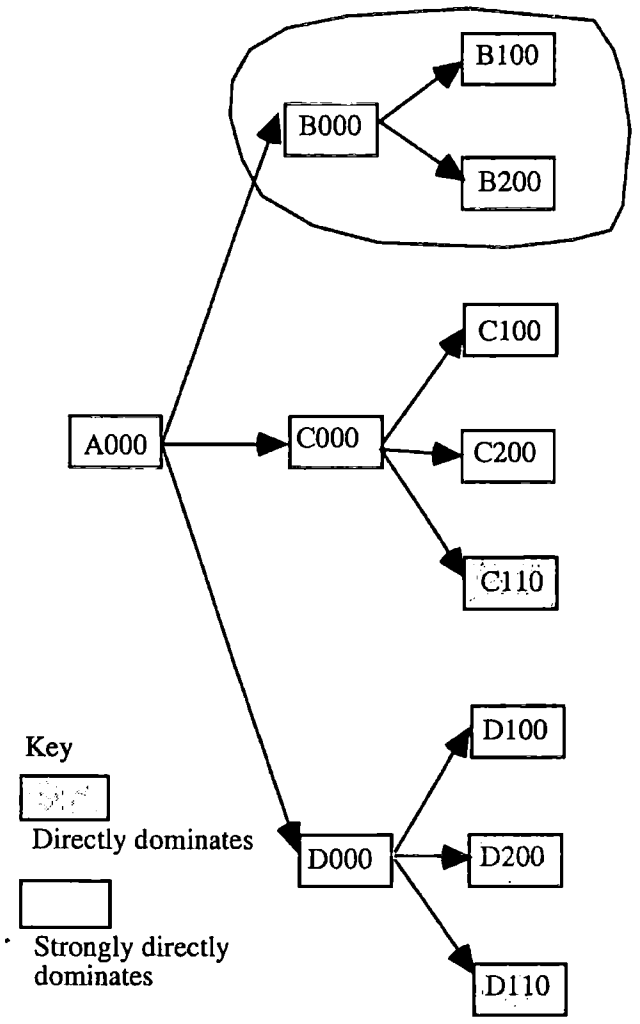


Figure 4.13: Showing only strongly dominant graphs selected.

Normally this process identifies and separates those subtrees which are strongly dominant. The direct dominant relations are, at least initially, rejected as unsuitable for reuse candidates, although SECTION partitioning (Step 10) may help to reduce the number of data inter-relationships. In Figure 4.13 the example from Figure 4.4 is shown indicating the subtree which would be selected by this step to form a reuse candidate. The other subtrees that were grouped within Figure 4.4 (but contain direct dominant nodes) may be again considered in the later stages of this method.

The result of this step is a number of reuse candidates. Before a decision can be made as to whether the units are to be reengineered for reuse, the following step on data slicing may provide some assistance towards reducing the workload required to encapsulate the reuse candidate.

4.9. Step 9. Identify Data Items in Reuse Candidates that would Reduce Data Interactions

4.9.1 Step Objective

- The objective of this step is to identify where the definition and usage of data items occurs across SECTIONs within different reuse candidates.

4.9.2 Approach

This step consists of two tasks. These are:

a) Identify types of interface for each of the reuse candidates
b) Reduce interfaces by removing or reducing specific sets of relations.

This step investigates the relocation of the data items where prolonged maintenance has unnecessarily dispersed data definitions and usages across different SECTIONs or reuse candidates. The consequence of carrying out this action is a reduction of the number of intersecting data items between reuse candidates.

a) Identify types of interface for each of the reuse candidates

Data interactions were identified in Step 5. By categorising data items using data item usages (i.e. Creation, Read, Updated and Deleted) unnecessary interactions between reuse units' data accesses can be identified. For instance, those SECTIONs where data items are initialized but not used should be identified and such initialisations moved to the data usage SECTIONs. The process involves the identification of a number of specific data relationships. The relationships are investigated as input and output parameters of the reuse candidates.

For each reuse candidate data item it is possible to define both input and output parameters. In general, the R (Read) relationships within a reuse candidate will indicate that this should be passed to the reuse candidate as an input parameter. The U (Update) relationship within a reuse candidate will indicate that it should be passed by the reuse candidate as an output parameter. Thus, the relationships defined within Step 5 are used within this step to investigate necessary external parameters to the reuse candidates.

Based on the input / output parameters of the reuse candidates, it is possible to derive a set of relationships between the candidates for each data item. These are expressed in terms of the operation

type for each data item between two candidates. Thus, a data item which is only read by the first candidate but which is read and updated by the other is expressed (R,RU)

b) Reduce interfaces by removing or reducing specific sets of relations.

If the basic relationships were derived directly from the data usages identified within Step 5 the interaction between the data items would appear to be very complex. For reuse candidates this is undesirable. However, there are some simplifications that can be applied to both input and output parameters to reduce their overall number. These are now described.

The parameter simplification process can be initiated by looking at specific relationships based on the results of Step 5. For instance, the (R, R) relationship means that the data items are not updated. If this is the case throughout the entire COBOL module, these can be considered more like constants and, therefore, in many cases, avoid the necessity of passing them as parameters. Furthermore, the (U,U) operations may be instances of local variables as each module will update the data item and ignore the operations of the other candidate. Thus in the (R, R) relationship there is no need to input data to either reuse candidate and also with the (U, U) relationship there is no need for either candidate to output the value of the data item for the other candidate.

In the examples above an opportunity to reduce both input and output parameters between two reuse candidates has been given. In many cases, it is not possible to reduce the data interactions between both input or output of each candidate, but still possible to reduce the interaction to a single direction. When a candidate only carries out an update operation on a specific data item, there is no point in the other reuse candidate out-putting data that will remain unread. Therefore, any single 'U' operation i.e. (U, R), (R, U), (U, RU) (RU,U) means that the other reuse candidate need not provide an output value.

An example will now be considered.

Data Item	In C1?	Usage	In C2?	Usage
INPUT-YEAR	1	R	1	R
INPUT-MONTH	1	U	1	U
INPUT-DAY	1	RU	1	U

Table 4.3: An example interface

In the example taken from Step 5 only those data items within the interface are included. Thus, this excludes TMP-C1 and TMP-C2. It is not necessary to consider those data items within this step.

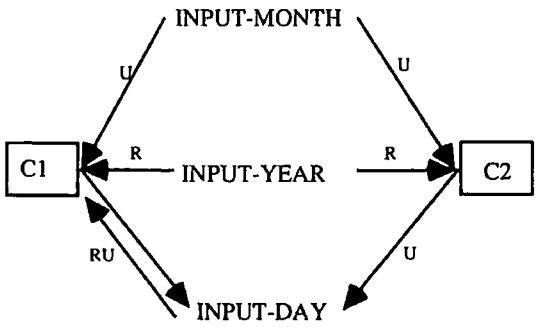


Figure 4.14: Input and Output of each data item to the reuse candidates

The results are represented graphically in figure 4.14. In the figure, the arrows to the reuse candidate represent passing of a value of that data item to the reuse candidate. Arrows pointing from the reuse candidate represent the passing of data from the reuse candidate when the value of the data item has been updated.

The long term benefit of making the reuse reengineering changes can be identified by assessing the reduction in the number of data interactions between SECTIONs.

The expected result of performing this step is an overall reduction in the interaction of data items between reuse candidates. The benefit of this step is a reduction in the complexity of their interface.

4.10. Step 10. Identify SECTIONs where Slicing could Assist Separation

4.10.1 Step Objective

- The objective of this step is to identify SECTIONs which implement more than one functionality.

4.10.2 Approach

This step consists of three tasks. These are:

a) Identify SECTIONs suitable for slicing.
b) Carry out detailed analysis to investigate if slicing is feasible.
c) Redevelop if feasible.

These tasks are now described.

a) Identify SECTIONs suitable for slicing.

Slicing the SECTIONs will benefit reuse by making the process of encapsulation of reuse candidates less complex. Furthermore, in some cases, it may mean that larger reuse candidates may be identified. Figure 4.15 shows the process of splitting to partition a SECTION (the SECTIONs are represented as boxes) which implements more than one functionality. This is achieved by slicing SECTIONs within cycles or those SECTIONs with a high degree of connectivity. SECTIONs with a high connectivity are those which PERFORM and are PERFORMed by a large number of other SECTIONs.

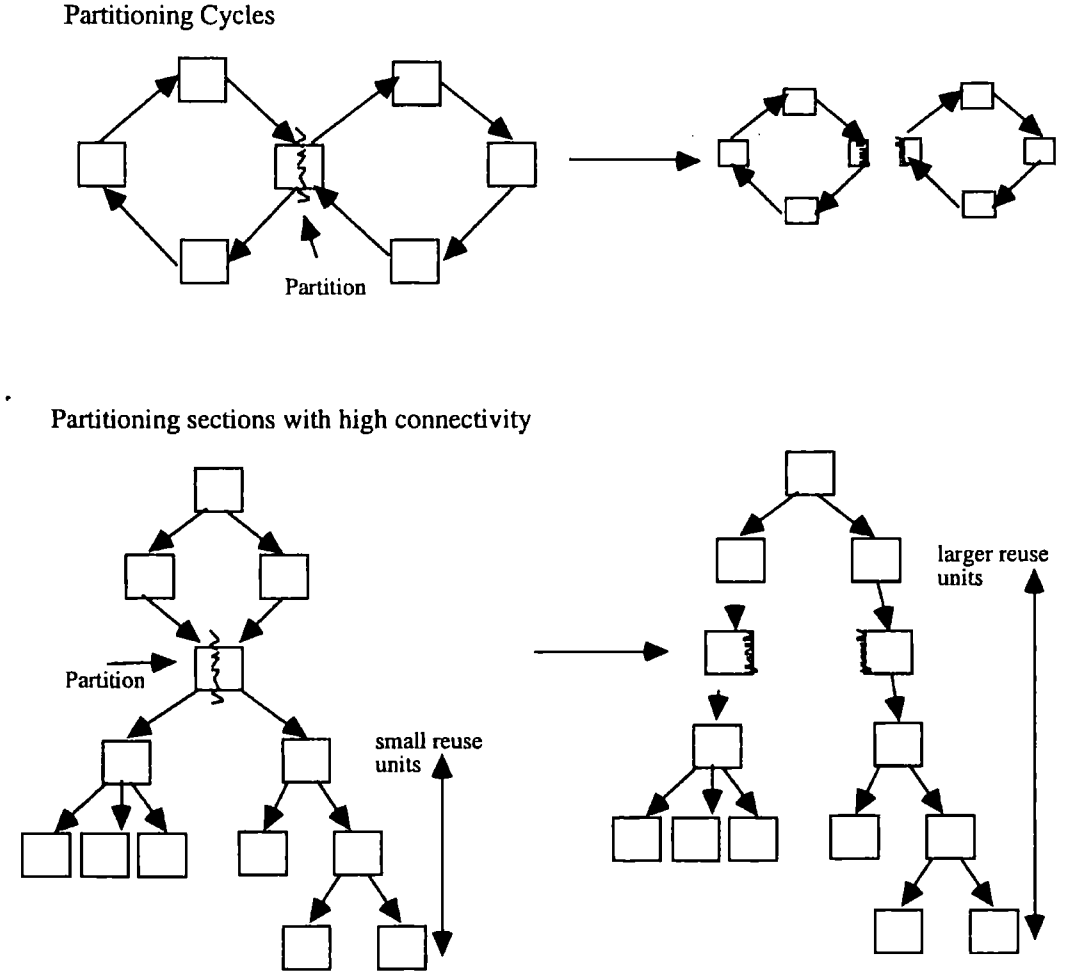


Figure 4.15: Partitioning SECTIONs

The expected result of this step is yet a further reduction in the complexity of a reuse candidate's interface. This is achieved, in particular, through the reduction in the number of shared data items between SECTIONs.

The slicing of SECTIONs is a useful approach for the identification of more, or larger, reuse candidates and also for the removal of cycles within the code. When cycles are removed the CDG becomes an ACDG. The usual approach to removing cycles is to produce a new 'grouped-node' where the SECTIONs within the cycle are represented. The disadvantage of this approach is that the grouped node inevitable becomes a 'hot spot' within the graph for an area of high fan-in and fan-out. If, at this stage during the method, the grouped node can be shown to be reducing the size of the reuse components available, then it may be appropriate to consider restructuring to remove the cycle.

b) Carry out detailed analysis to investigate if slicing is feasible

Two approaches can be applied towards the partitioning of SECTIONs. These are:

- Subgraph of data - graphs of data dependencies are generated and the interconnection of the resulting graph is analysed to see if there are any disconnected subgraphs.
- Ranking - the usage of data items are totalled for each reuse candidate and, based upon this usage, data items for re-location are proposed.

Subgraphs of data are composed of the data dependencies between the data items within the SECTION. For instance, a graph would be constructed as shown in Figure 4.16:

COBOL Statement	Graphical Result
MOVE X TO Y	
ADD Z TO X	
SUBTRACT Y FROM Z	

Figure 4.16: COBOL Statements and their graphical representations

A graph (Figure 4.17) can then be constructed to show the interrelationships between the individual data items.

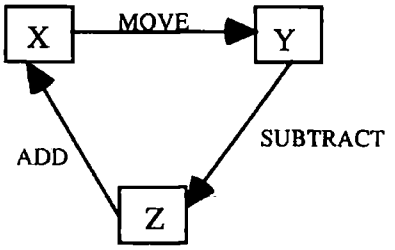


Figure 4.17: The Data Dependency Graph

From the full analysis of the SECTION, a number of subgraphs (data items sets which, within the SECTION, operate independently of the sets) may be identifiable. For instance, a further set of data items, A, B and C may be found to be connected to one another but not to the set X, Y and Z. In more complex examples detailed data flow analysis may be required.

If subsets of data are identified, then the callee and called SECTIONs of the SECTION being analysed and the callee output parameters and the input parameters of the called SECTIONs should be considered. Hopefully, examples such as the following will be found.

Given a SECTION PERFORMed by at least two other SECTIONs and which itself PERFORMs at least two SECTIONs it may be possible to slice the SECTION. This process is similar to the case of partitioning of cycles. To demonstrate this principle the following example is used.

The example assumes the following

The callee PERFORMs of the SECTION to be partitioned are P1 and P2.

The called SECTIONs are referred to as P3 and P4.

The data items used within the SECTION are X, Y, Z, A, B and C.

If P1 is shown (within Step 9) to be inputting X and Z, and P2 to be inputting A then it would appear that the two PERFORMing SECTIONs are using different data sets.

To further the analysis process, it is important to establish that the same is true of the output via the PERFORMed SECTIONs. If P3 were outputting Y and Z and P4 were outputting A and B, then it is also the case that the two PERFORMed SECTIONs are also outputting different data sets. Thus, it is assumed that if two distinct sets of data are used, then the SECTION carries out two functions and therefore there would be a benefit from the partitioning of the SECTION. Given this case, depending on the remaining logic of the SECTION, it may be possible to slice the SECTION. This process is shown in Figure 4.18.

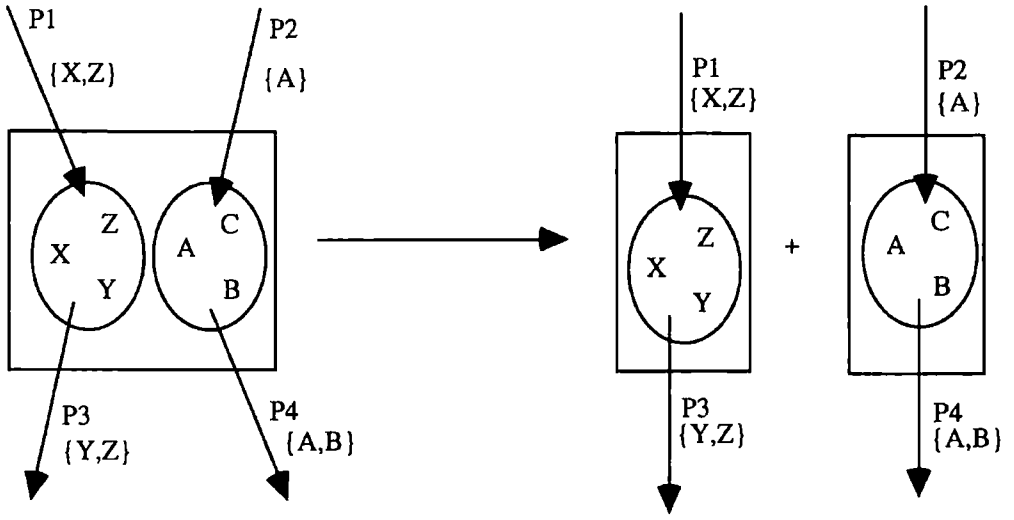


Figure 4.18: The Process of Splitting a SECTION

In general this process is similar to the data clustering used for the identification of candidate reuse units. In this case however, the clustering approach is restricted to a specific SECTION. Like the clustering approach used within Step 8, obtaining such a clear and simple separation of data is not always possible. In this case, additional reengineering must be applied to obtain separation. The cost implication for this additional reengineering work, must be considered and compared with the benefits that will be obtained.

Another approach that may be considered, is the re-location of data items. To identify appropriate data items for re-location a ranking process is recommended. The ranking process is achieved by comparing usage of each data item within each reuse candidate. The total usage of the data items are then ranked using the following formula.

Rank value = (Total usage within candidate -
Total usage within all other candidates)

Data items that rank highly within one reuse candidate but achieve a low position within the other candidates have potential for re-location. Specifically, those data items which when ranked appear within the top third are those that are reviewed as feasible for relocation. Any candidate with low usage of the data item should be evaluated for re-location potential, and the high usage candidate should be evaluated for the receipt of the data item.

Examples where the relocation is feasible, and often requiring minimal reengineering, are the creation of data items and where they are not used further within the SECTION. The identification of the creation of data items is addressed within Step 5.

c) Redevelop if feasible

If the process shows the slicing to be feasible then the SECTION can be partitioned into two separate SECTIONs. Thus allowing larger candidates to be obtained by the earlier steps of this method. The disadvantage being that it will be necessary to repeat some of these steps in order to establish the consequences of the change.

4.11. Summary

This chapter has described a method for identifying reuse candidates from legacy COBOL systems. The method consists of 10 steps that should be carried out in sequence, however in some cases it may be necessary to repeat earlier stages. The approach includes a variety of sources of system support information to aid the reuse candidate identification process. For instance, source code and user and technical documentation are used along with 'expert' assistance from the users and designers of the code. However, the method relies heavily upon the source code as it is recognised for old and highly maintained systems that this may be the only available or only accurate source of information.

The following chapter defines some requirements for tool support to assist the maintainer carry out the steps described above. Chapter 6 takes the method and investigates its use by describing a number of case studies.

Chapter 5. Tool Support

This chapter introduces the requirements for tool to support the use of the method. Existing tools that are used to demonstrate the steps are overviewed and some of their deficiencies are identified. The existing tools in most cases have been sufficient to demonstrate the principles of the method but are inadequate for commercial exploitation. The existing tools are however, useful for the derivation of requirements for a full tool set and are summarised below.

5.1. General Support Tools

In order to support this reuse reengineering method a number of support tools are required. Some of the tools are specific to one or more step within the method, others can be seen as more general and supporting the method as a whole. The requirements and a description of current level of support provision for the specific tools will be given for the individual steps in the following section (Section 5.2). The requirements for the more general tools are given below:

1) A code analysis tool. Specifically, a COBOL analyser that provides details on the PERFORM structure and SECTION data usage.
2) The provision of some basic reengineering functions to restructure code where pre-conditions to the method do not apply. For instance, the ability to re-structure the code to remove GOTOs.
3) A tool to support slicing of the code. This should be general in that slicing may be performed at the SECTION or data level and the slicing criteria used will vary depending on the current step.
4) A visualisation tool is required that displays nodes and edges and allows their positioning to be manipulated. The selective use of colour to highlight specific nodes if also useful as is the ability to group nodes. The visualisation tool should where possible ensure constant positioning of nodes throughout method for comparisons to be easily made.

The tool should require little user experience, it should also be fully integrated. Additionally, it should be capable of supporting large commercial applications.

5.2. Tool Support for Method Steps

This section investigates the requirements and current level of tool provision for each of the steps of the reuse reengineering method.

5.2.1 Tool Support for Step 1

The code analysis tool is used to:

1) Establish if pre-conditions for the application of the method apply. For instance, that there is no use of GOTOs between SECTIONs.
2) Identify the SECTIONs which the module and the PERFORMs between each of the SECTIONs.

The visualisation tool is used to:

1) Generate the PERFORM graph representing each SECTION as a node and each unique PERFORM as an edge. Multiple PERFORMs between two SECTIONs should be represented as the total number of multiple PERFORMs as a label on the edge.
2) Layout the graph to ensure that it is clearly presented. For instance, no nodes overlapping and as fewer edges crossing as possible.

The PERFORM graph is generated by identifying the PERFORM relationships between SECTIONs. However, before accurate PERFORM graph analysis can be assessed a number of conditions, which are feasible within the COBOL language must be satisfied. If any of the conditions are present restructuring of the code must be completed before PERFORM analysis commences. These changes can be carried out by the restructuring tool.

The PERFORM relationships are identified using simple pattern matching approaches such as grep. This simple approach identifies all PERFORM statements which include inline PERFORM statements. These need to be removed manually as contextual information is required that is not available to simple tools such as grep. The relationships are then converted to the input format of the graphical display tool.

5.2.2 Tool Support for Step 2

Tool support for Step 2 should:

1) Identify cycles and remove them from the PERFORM graph. This should be achieved by grouping the nodes within a cycle into a single node.
2) Identify the strong and direct dominance relations.

The visualisation tool is used to:

1) Provide a visual representation of the dominance tree. The strong dominance nodes should be shaded. The edges should be labelled 'S' for strong and 'D' for direct dominance. This should enable visual identification of the relations.
2) Provide a tree layout algorithm.

The dominance tree is based on the PERFORM relationships thus the (call : callee) relationships from Step 1 are used. The dominance tree is then generated by identifying all possible paths through the PERFORM graph from its entry point to each leaf node. The entry point is the initial starting point of the code (the SECTION at the beginning of the PROCEDURE DIVISION).

The dominance tree tool is currently implemented in Prolog. Cycles, if they are found, are grouped into a single new node to obtain an acyclic graph. The output of the Prolog is then transferred to a C program. The program uses the following algorithm to generate the dominance tree:

- take 3 nodes of the paths obtained as above, in the form for instance X, Y and Z (when X PERFORMs Y and Y PERFORMs Z)
- if all paths from node X to node Z pass through node Y, then Y strongly dominates Z
- if other paths exist between from X to Z and do not pass through Y, then X directly dominates Z

The output of the program is then used as the input to the graphical display tool.

5.2.3 Tool Support for Step 3

Tool support for Step 3 should:

1) Identify subtrees within the dominance tree.
2) For each of the subtrees identified, select those consisting of only strongly dominant nodes.

3) For each subtree with only strongly dominant nodes identified, investigate if lower level subtree is present.
4) Visually represent the subtrees by circling them on the dominance tree visualisations.

This analysis process is generally performed visually. This process is relatively simple using the graphical representations. For textual formats the identification process can be performed automatically. In this case it is necessary to identify each subgraph and investigate the types of dominance relations that are present.

5.2.4 Tool Support for Step 4

The code analysis tool is used to:

1) Identify the data items accessed for each SECTION.
2) Identify, for each data item, whether it is read and / or updated.
3) Categorise data base items with two additional type (creation and deletion).
4) Identify sets of GROUP items from WORKING STORAGE and their corresponding ELEMENTARY data items.
5) Identify relationships between ELEMENTARY and GROUP data items for each SECTION. For instance, a read to an ELEMENTARY item within one SECTION and an update for the ELEMENTARY item's GROUP item within another SECTION.

The provision of tool support for this step is beyond the scope of the small scale prototype tools that have been used so far within this method. Due to the analysis requiring knowledge of the way in which every data item is used tools would need to understand the syntax of COBOL. There are a number of tools on the market that would be capable of the provision of such information and therefore the development of such tools within this work cannot be justified. Within the case study the analysis work was conducted manually using simple scanning tools.

The provision of graphical representations such as the 'consists' relationship is generated using another binary relation and converted for graphical display as in Steps 1 and 2.

5.2.5 Tool Support for Step 5

Tool support for Step 5 should:

1) Identify areas for data independence using the data inter-relationships identified within Step 4.
2) Apply cluster analysis using Jaccard's Coefficient for each of the subtrees identified in Step 3.
3) Visualise the results of the previous points and also provide a summary of the results in a tabular form. This should indicate:
    - The total number of data items
    - The total number of data items within a specific subtree
    - Data item unique to the subgraph i.e. those not read or updated externally of the subgraph
    - Data items which are not unique to the subgraph i.e. those that are read and / or updated within and externally of the subgraph. These are termed the candidate's data interface.
4) For each data item in the candidate's data interface use the results of Step 4 to identify the data items which can be excluded from the interface.

Currently, the data necessary to support the cluster analysis, used within this Step, is input into a spreadsheet. The interactions between data items are too numerous and complex to represent in their entirety in a graphical form. Thus, spreadsheets are used to analyse interactions and provide a flexible textual representation. Further support tools are required to allow for differing clustering algorithms to be selected.

5.2.6 Tool Support for Step 6

Strictly no tool support is necessary for carrying out this Step, however, in some cases the experts may be assisted by tools:

1) That allow arbitrary groupings of data items / SECTIONs to describe sets of functionality identified by the experts.
2) That automatically identify and visualise differences in the automatically identified candidate reuse units and those identified by the experts.

No tool support is currently available for this Step.

5.2.7 Tool Support for Step 7

Tool support for Step 7 should:

1) Provide a means to support graph analysis and simplification, including:
    - identification of areas of high fan-in

- the temporary exclusion of specific SECTIONs from the graph
- identification of data items not related to a specific functionality i.e. non-functional requirements
- temporary exclusion of data items not updated through entire code module
- temporary exclusion of data items used externally of the code

2) Assist document analysis to help the identification of non-functional requirements and management of lists indicating those sets of non-functional requirements used most frequently within the code.

Three different type of tools are used within this step. These are:

- document analysis – this includes support for the location of noun and verb phrases for the identification of concepts.
- graph analysis – for instance, for the identification of specific characteristics of the graphs (i.e. areas of high fan-out).
- graph simplification – techniques to allow the temporary exclusion of specific sets of nodes and edges from the graph or the ability to slice on a specific criterion.

5.2.8 Tool Support for Step 8

Tool support for Step 8 should:

1) Identify all information relevant to the potential reusability of the candidate. This information should be collected and presented to the user.
2) The exact composition of the reuse candidates should be presented. This is achieved by graph slicing.
3) A visual representation of the reuse candidates should be generated, showing PERFORMs and data interactions.

The slicing tools (also used in Step 7) are used to perform some of the basic slicing techniques on the graphical representations. No tools are available at present for the slicing at the code level.

5.2.9 Tool Support for Step 9

Tool support for Step 9 should:

1) Identify the actual interface for each reuse candidate

2) It should be possible to show the step-by-step process of removing data relations i.e. the removal of read only data items (these are removed since these data items represent constants within the program)
3) Provide visualisation and tabulation forms of results.

At present there is no automated tool support for points 1 and 2 of this step. The provision of automatic graphical representation of input and output parameters would be useful. Support for this step could be provided easily given the data generated from other steps, such as the data usages.

5.2.10 Tool Support for Step 10

Tool support for Step 10 should:

1) Provide a list of SECTIONs that may benefit from splitting.
2) Perform detailed analysis to investigate if splitting is feasible using slicing and clustering techniques.
3) Automatically transform SECTIONs into two independent SECTIONs and update the code (i.e. PERFORMs).
4) Provide tabular and graphical forms of the subsets of data and areas where data item usage overlap (the reuse candidate data interface).

No tools support is available for points 1 to 3 of this step. In addition, the generation of tools support would require detailed knowledge of the COBOL syntax as well as slicing procedures possibly including some dynamic data analysis.

5.3. Summary

This chapter has identified the need for four general tools to support the use of the method. These are a COBOL analysis tool, a restructuring tool, a code slicer, and a graphical display tool. Numerous other special purpose tools are also required to support individual steps within the method. For instance, these include document analysis tools and graph simplification tools. Ideally, the tools to support the individual steps within the method should be fully integrated so that it is unnecessary for the user to import and export the results of each step. However, it would be beneficial if the general support tools were independent. For instance, the users could select their own graphical display tool. This would be particularly beneficial with the code analyser as other languages could then be supported by the method. In addition, the ability of the user to select tools with which he is familiar will ensure a shorter learning time for the use of the tools.

Chapter 6. The Case Studies

This chapter describes the results that were obtained from applying the method described in Chapter 4 to real world problems. The results shown in this chapter are of actual examples taken from industrial information systems.

For the purpose of assessing the feasibility of identifying reusable code, a COBOL application is used. The application consists of nearly 4 million lines of code divided into about 3,000 on-line and batch programs. Many of its individual component systems link to databases, some of which are larger than 600 gigabytes and have access to up to 300 million records. The code has been annotated using Jackson Structured notation at the SECTION level. Many of these systems have been highly maintained and the original structure of the code has been lost.

In order to describe the results of applying the method described in Chapter 4, examples are provided from three samples of code. These samples will be referred to as case studies A, B and C. Case study A is the smallest code sample examined. It comprises 440 lines of code consisting of 12 SECTIONs and 120 data items. Case study B is of medium size and is an example of COBOL code designed to access a large database. The module consists of 21,000 lines of code, 221 SECTIONs and over 4000 data items. Finally, case study C is the largest code sample analysed. This consists of about 40,000 lines of code split into 230 SECTIONs and containing 2823 data items.

The three case study COBOL samples have been selected for analysis by virtue of the fact that the samples are fairly typical of the COBOL studied. Some of the samples were better documented and some contained many divergences from standard formatting and naming conventions. However, in general, each of the samples is well written and well structured. For instance, SECTIONs are written in a procedural style. In most examples given in this chapter the names of the data items and SECTIONs have been changed to comply with confidentiality agreements.

This chapter uses the same format as for Chapter 4; the results of each stage of the method are described within a separate subsection. The tasks of each of the steps are detailed with the results obtained from their application. The results of the detailed case studies now follow.

6.1. Step 1. Generate a PERFORM Graph from the Source Code

- The objective of Step 1 is to identify the calling structure of the code on a SECTION basis.

a) Check preconditions for producing the PERFORM graph

For each of the three case studies it was found that the code was suitable for the graph to be generated without the necessity of making changes. For instance, there were EXIT statements at the end of each SECTION and no GO TOs between SECTIONs. Thus, the code is well structured.

b) Generate the PERFORM graph

The resulting PERFORM graph from the analysis of case study A is shown in Figure 6.1.

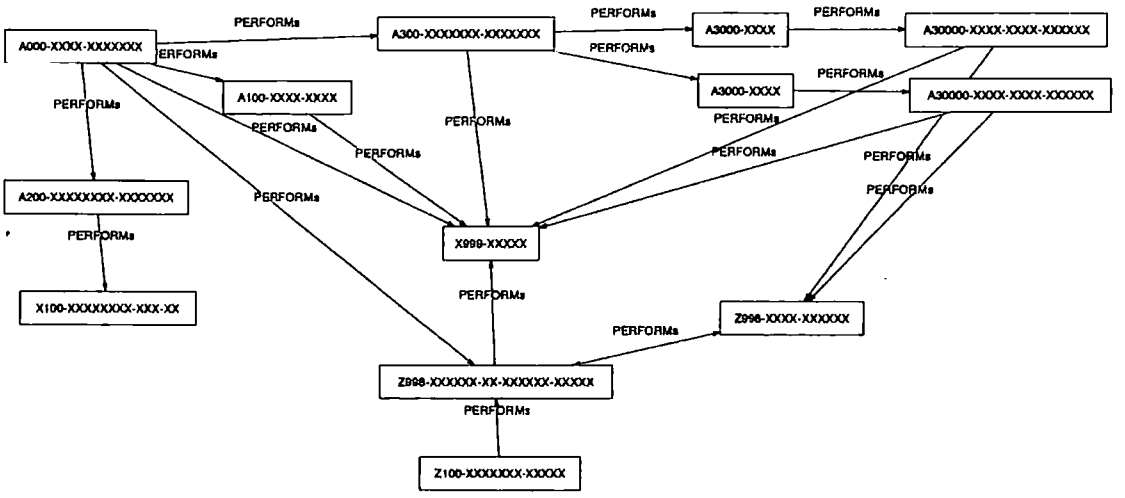


Figure 6.1: The PERFORM graph for case study A

A tool called GraphTool produces the graphs represented within this thesis. This is an in-house tool was developed at Durham by Thierry Bodhuin and later reengineered under the EPSRC RELEASE project by Pete Young.

The SECTIONs (represented by the nodes in the graph) are labelled with letters. These are replacements for the original names within the source code. The directed lines between the nodes represent the PERFORMs.

The resulting PERFORM graph from the analysis of case study B is shown in Figure 6.2 and that of case study C is shown in Figure 6.3.

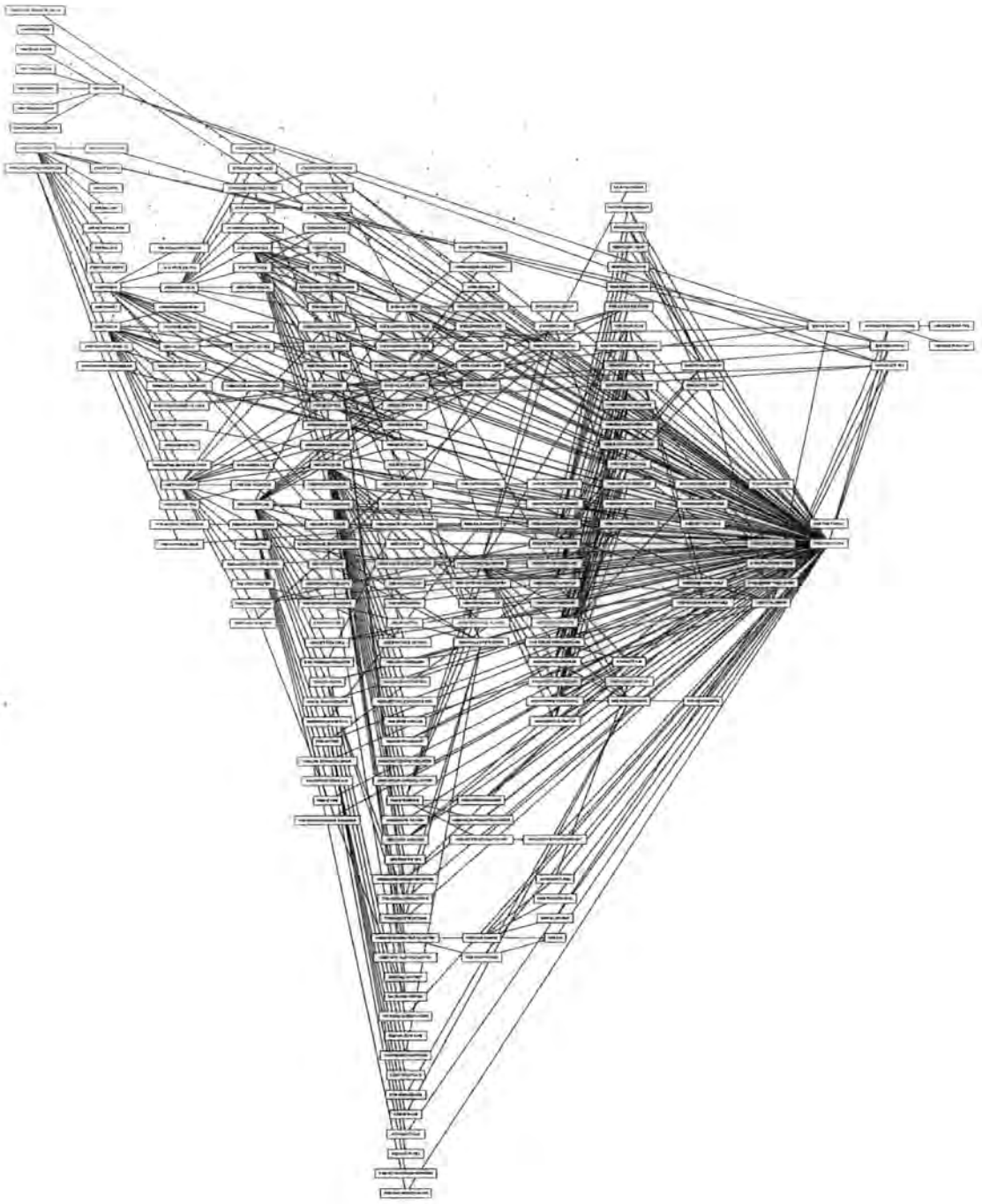


Figure 6.2: The PERFORM graph for case study B

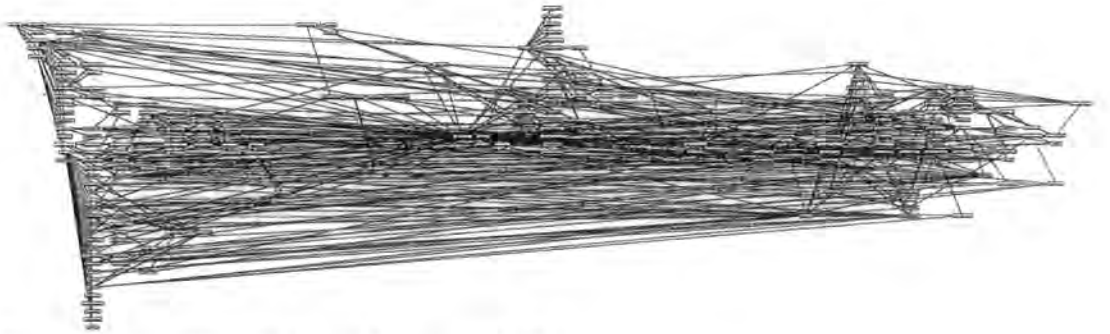


Figure 6.3: The PERFORM graph for case study C

The generation of the PERFORM structure of the code represents the completion of the first step of the method. The resulting graphs are quite complex (especially for the larger source code samples) due to the number of PERFORMs between SECTIONs as well as increased numbers of SECTIONs. In order to gain an indication of the potential candidate reuse units within the code a simpler representation needs to be sought. Reducing the multiple PERFORMs to simple arcs and representing the graph as a tree structure assists this process. Hence, the following step investigates the formation of a dominance tree.

6.2. Step 2. Generate a Dominance Tree from the PERFORM Graph

- The objective of this step is to identify the dominance relationships between SECTIONs.

a) Remove cycles from the PERFORM graph

Cycles must be removed from the PERFORM graph before a dominance tree can be generated. No cycles were found in the PERFORM graph for case study B. However, a single cycle consisting of two SECTIONs was found in case study A, and another cycle consisting of 62 SECTIONs was found for case study C.

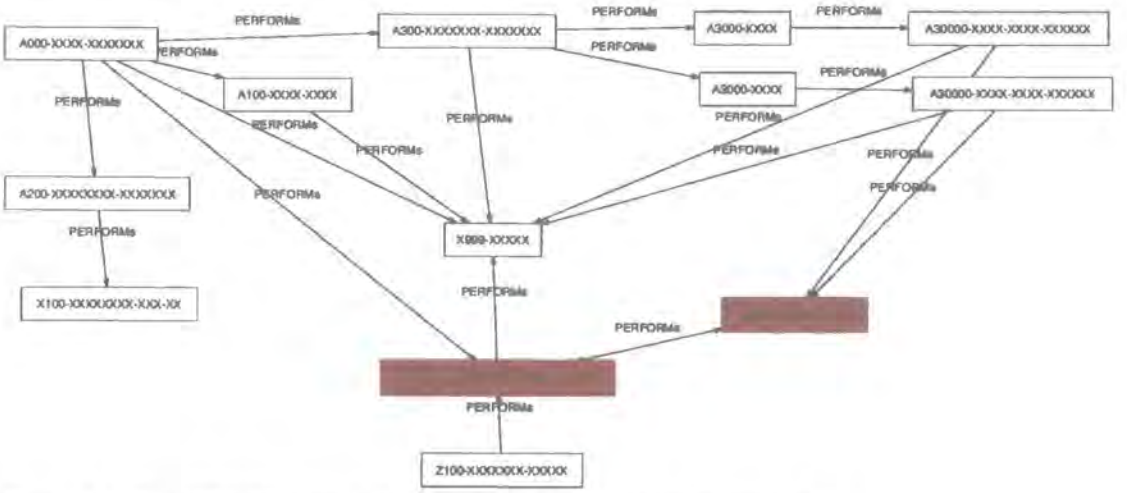


Figure 6.4: The PERFORM graph for case study A (nodes in cycle are shaded)

The nodes involved within the cycle should be treated as a single node. The nodes involved within a cycle for case study A are shaded in Figure 6.4. Thus, all PERFORMs to or from SECTIONs within the cycle are to the new node. The PERFORM graph should then be re-drawn with the new node. Figure 6.5 shows the new PERFORM graph.

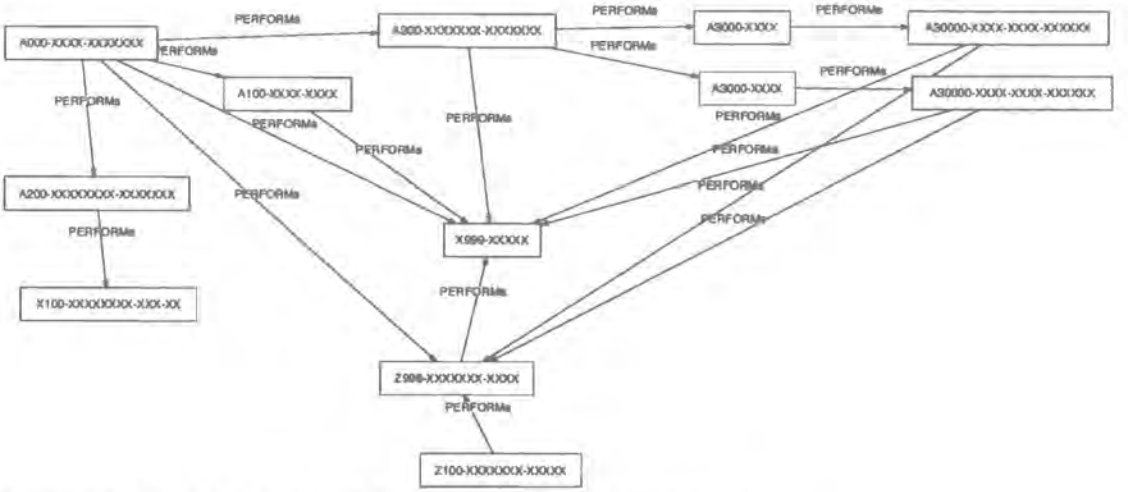


Figure 6.5: The PERFORM graph for case study A with grouped node

Those nodes involved within a cycle for case study C are shaded in Figure 6.6.

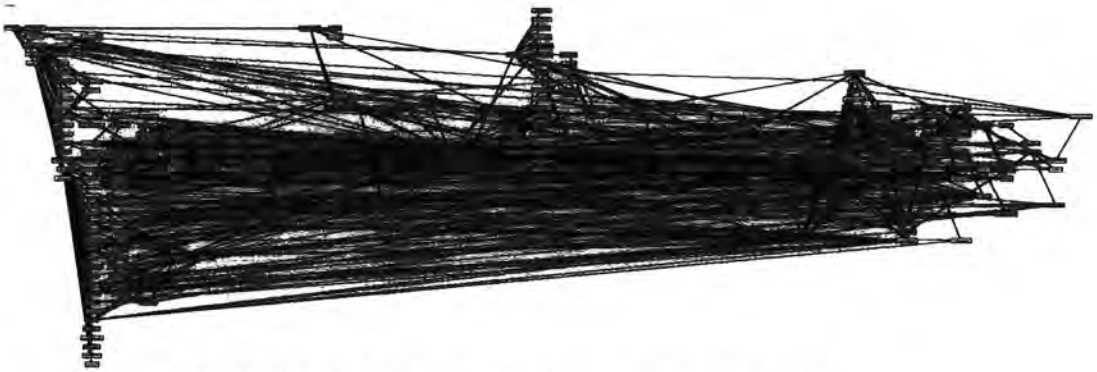


Figure 6.6: The PERFORM graph for case study C (nodes in cycle are shaded)

The resulting PERFORM graph with the grouped nodes is shown in Figure 6.7.

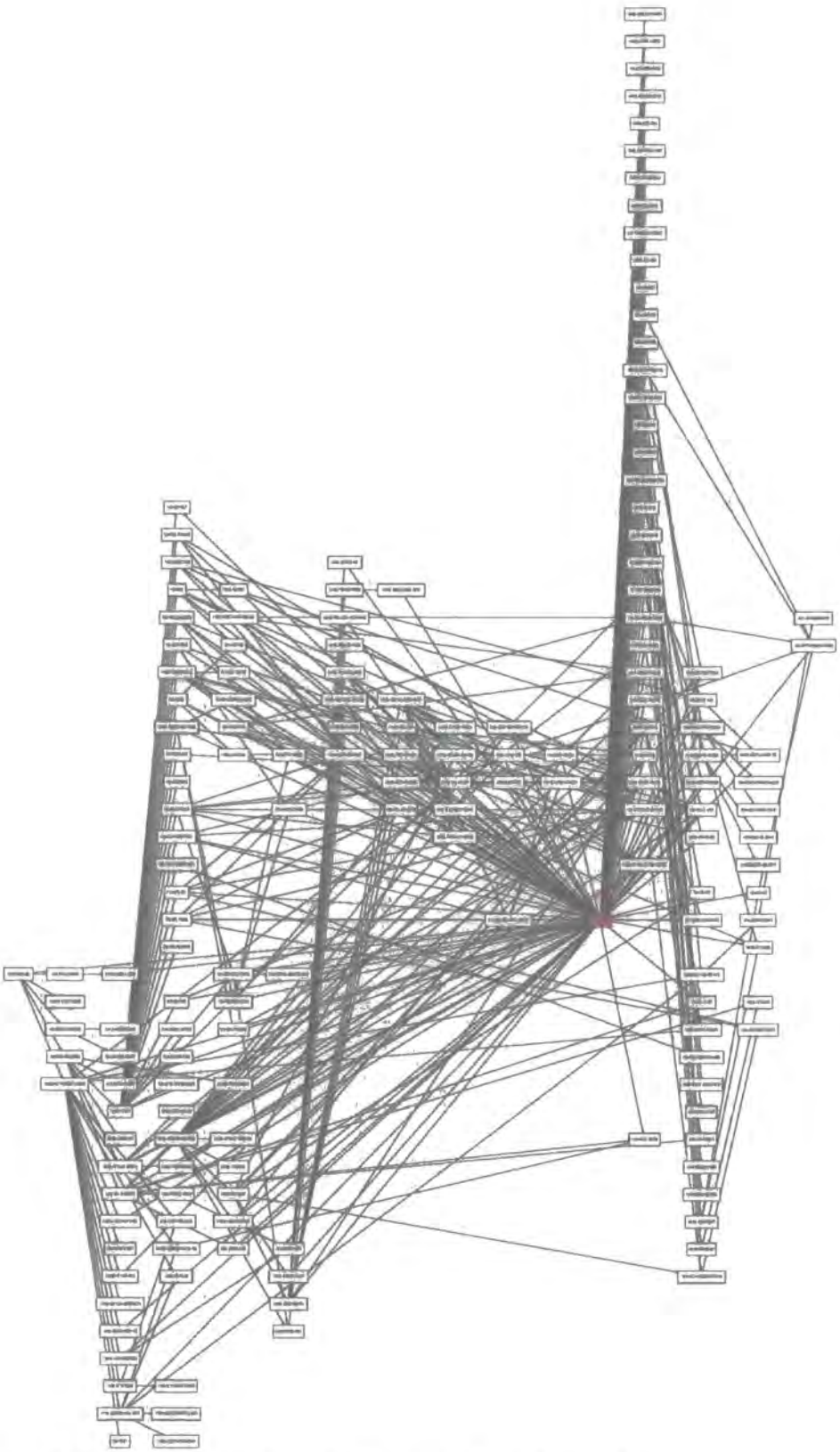


Figure 6.7: New PERFORM graph for case study C with grouped nodes

b) Generate the dominance relations

Once an acyclic PERFORM graph is generated, then the dominance tree can be obtained. The dominance tree gives an initial indication of potential candidates reuse units within the code module. The graphical representation of the dominance tree assists discussion between technical and non-technical (i.e. managerial) staff. The dominance tree for the PERFORM graph of case study A (Figure 6.5) is shown in Figure 6.8.

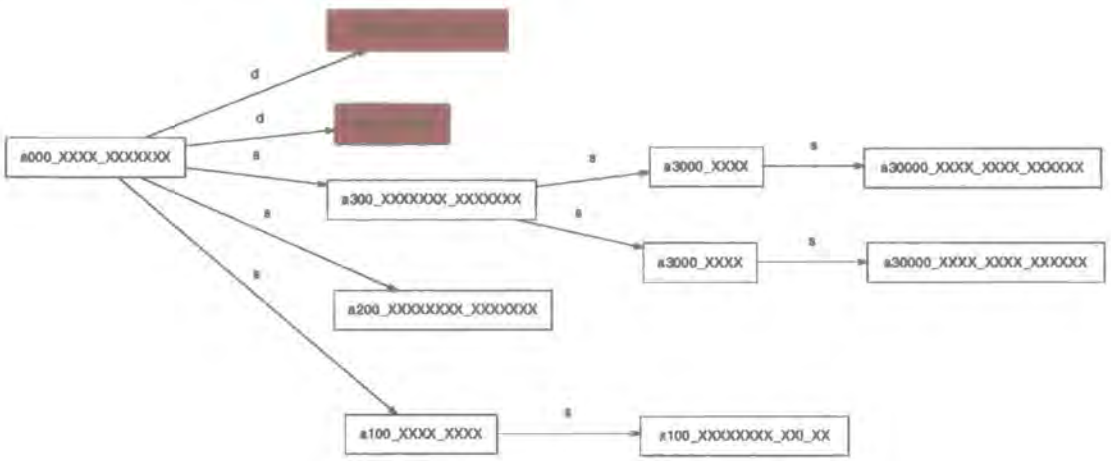


Figure 6.8: The dominance tree for case study A

In Figure 6.8 the node a000_XXXX_XXXXXXX directly dominates z996_grouped_node and x999_XXXXX and strongly dominates a300_XXXXXXX_XXXXXXX, A200_XXXXXXXX_XXXXXXX and a100_XXXX_XXXX. The strong dominance relations are represented with a 's'; the directly dominant relations with a 'd'.

The dominance tree for case study B is shown in Figure 6.9.

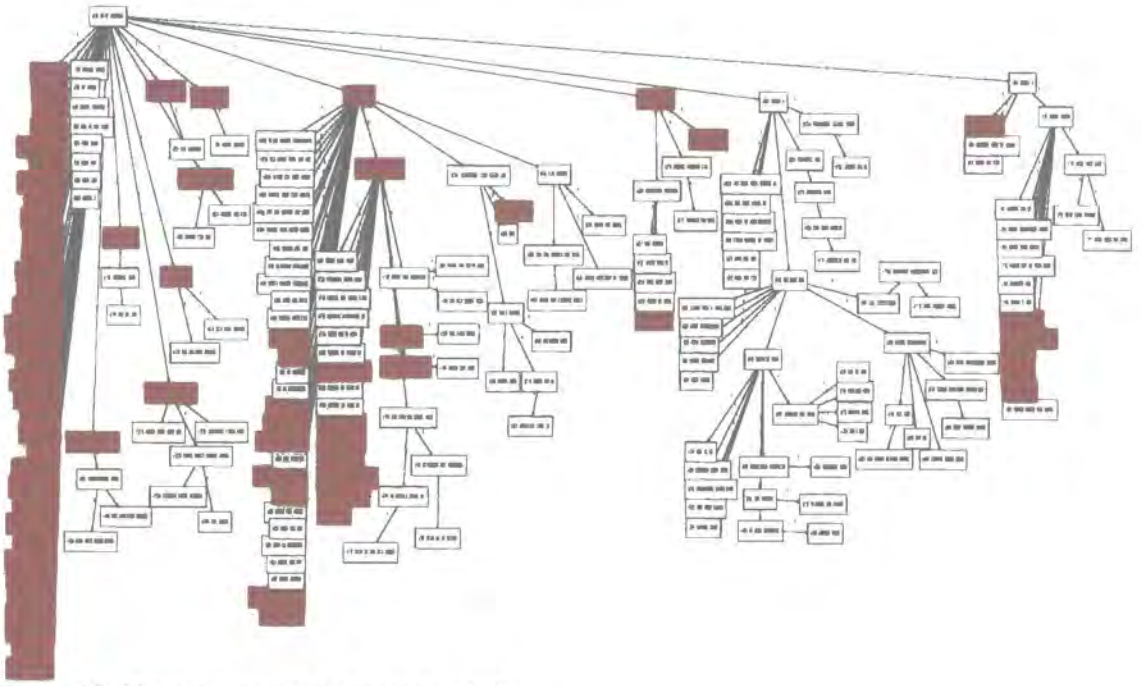


Figure 6.9: The dominance tree for case study B

The dominance tree for case study C using the acyclic graph is shown in Figure 6.10.

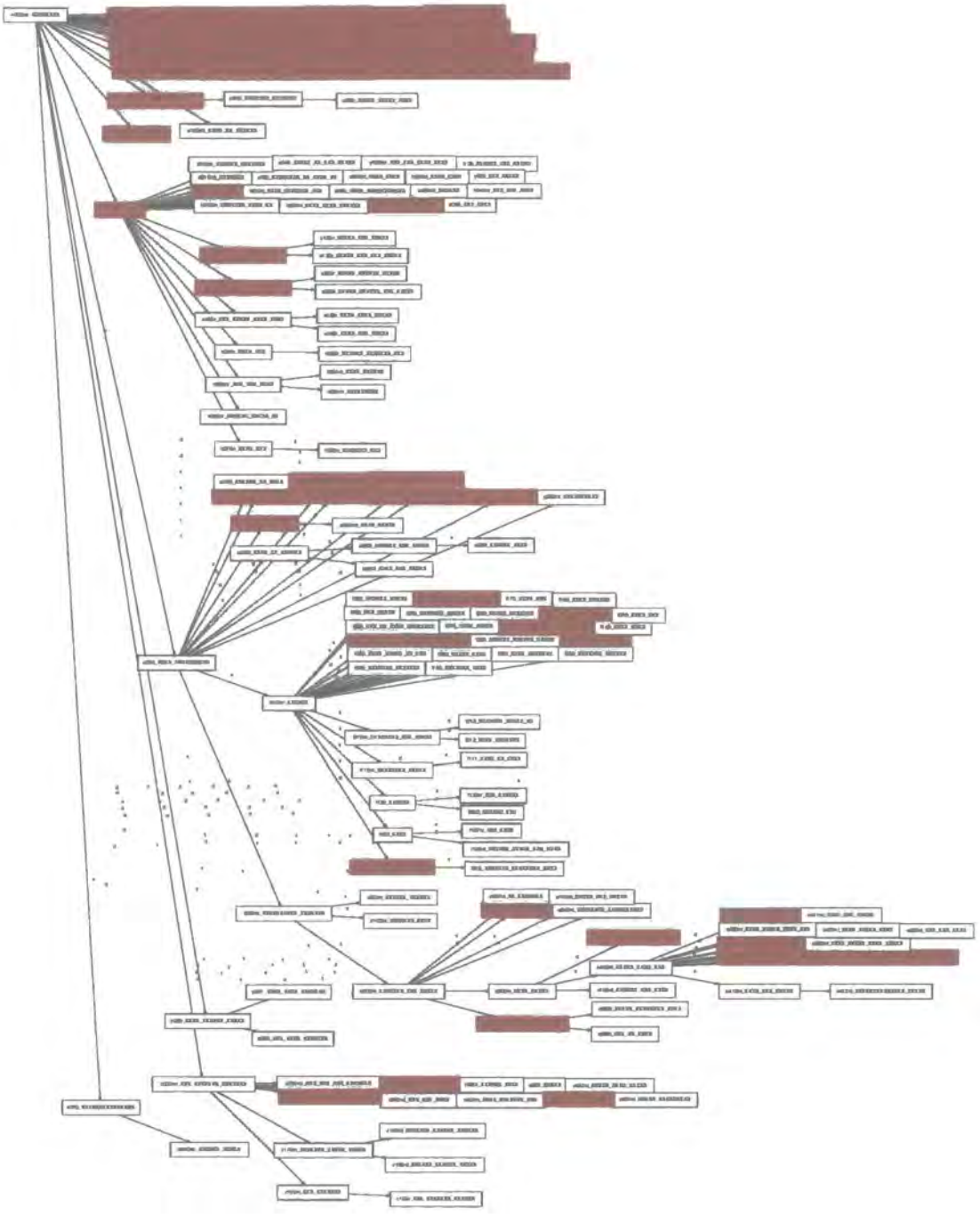


Figure 6.10: The dominance tree for case study C

Subtrees of the dominance tree can now be analysed using the two relations identified of strong and direct dominance. The direct dominance relations are the nodes that have been shaded within the

dominance trees. The subtree identification process (hence the identification of candidate reuse units) forms the basis of the following step.

6.3. Step 3. Identify Candidate Reuse Units from the Dominance Tree

- The objective of this step is to identify as many reuse candidate units as possible. In addition, units at different levels of granularity should be considered.

a) Identify candidates at varying levels of granularity

Using the dominance tree for each case study, it is now possible to identify a number of potential candidate reuse units. Locating subtrees containing a number of strongly dominant nodes can identify the potential candidate reuse units. In the figures 6.11, 6.12 and 6.13 the dominance trees generated from Step 2 are grouped into prospective candidates reuse units. The rough groupings based on the subtrees indicate possible candidates reuse units which can be obtained from the COBOL code. Those reuse candidates which contain the fewest directly dominant (shaded nodes) will be the easiest to reuse (due to their simpler PERFORM structure) as the PERFORMs are restricted to be within the candidate.

The potential candidates reuse units that can be obtained from the three case studies are indicated in the following annotated copies of the dominance tree. Figure 6.11 shows the candidates for case study A (from Figure 6.8)

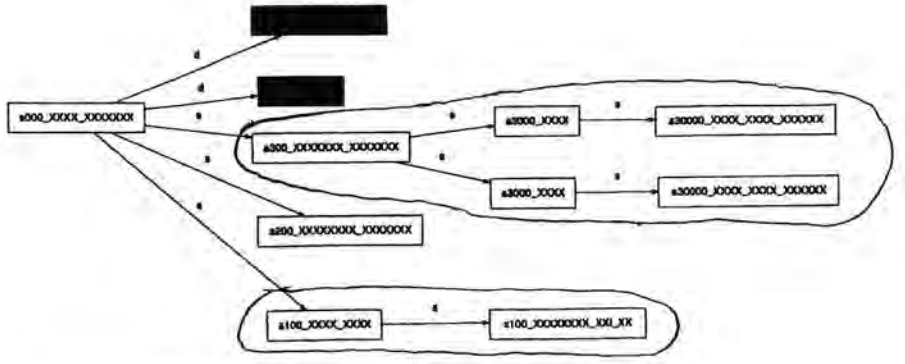


Figure 6.11: Reuse candidates from case study A

Figure 6.12 shows the candidates for case study B from Figure 6.9.

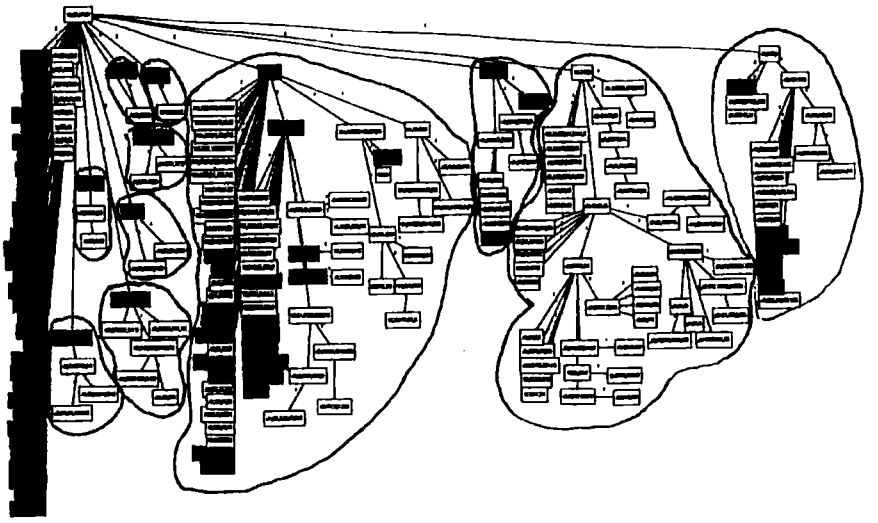


Figure 6.12: Reuse candidates from case study B

Figure 6.13 shows the candidates for case study C from Figure 6.10.

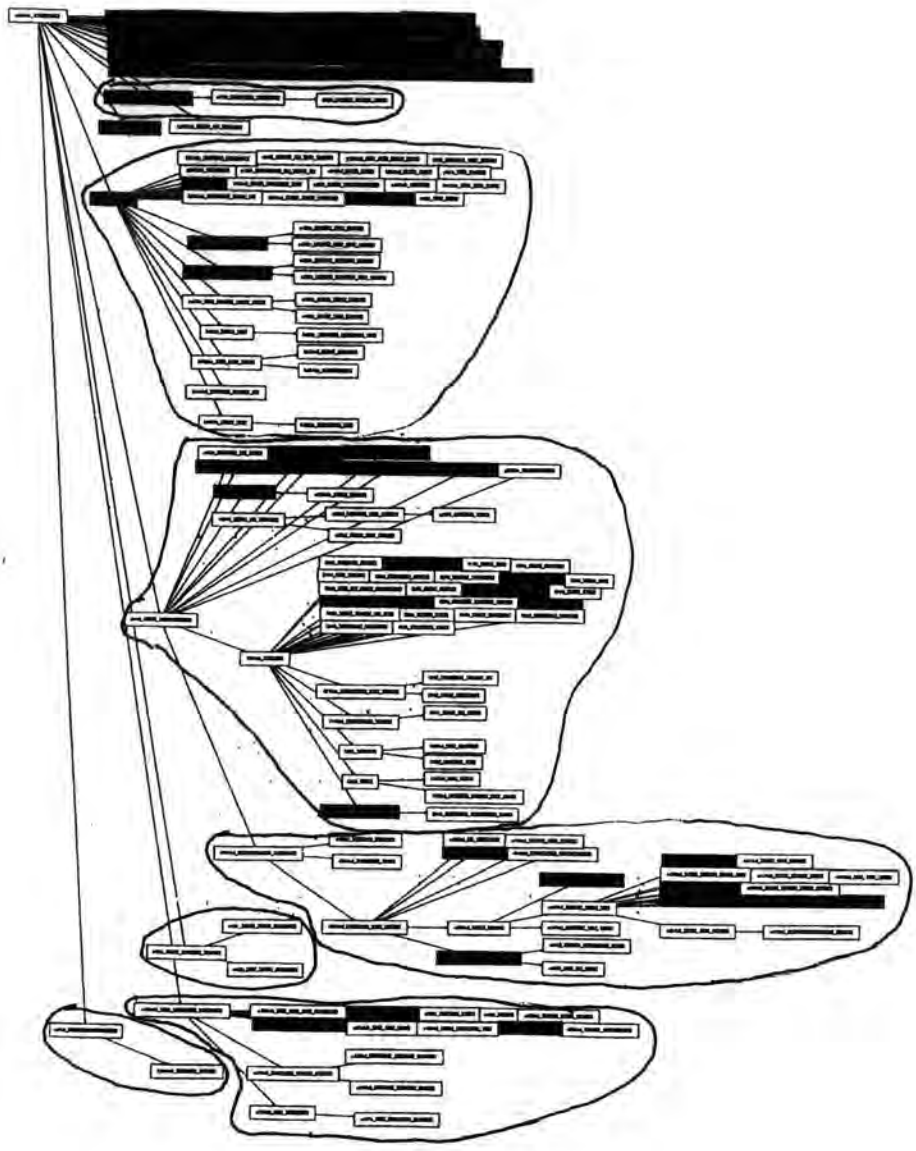


Figure 6.13: Reuse candidates from case study C

The subtrees can be identified at a number of levels. For instance, a single subtree could be decomposed into a number of smaller subtrees. Taking one of the highest-level candidate reuse units from case study B, a number of candidates reuse units at different levels of decomposition can be found. The subgroupings within an individual candidate are represented in Figure 6.14. In this figure the subtrees which were grouped within the initial candidate reuse unit are grouped using the same procedures as described above. So for example, d000_XXXX_X is one candidate reuse unit from Figure 6.12. This can then be divided up into 12 units shown within Figure 6.14.

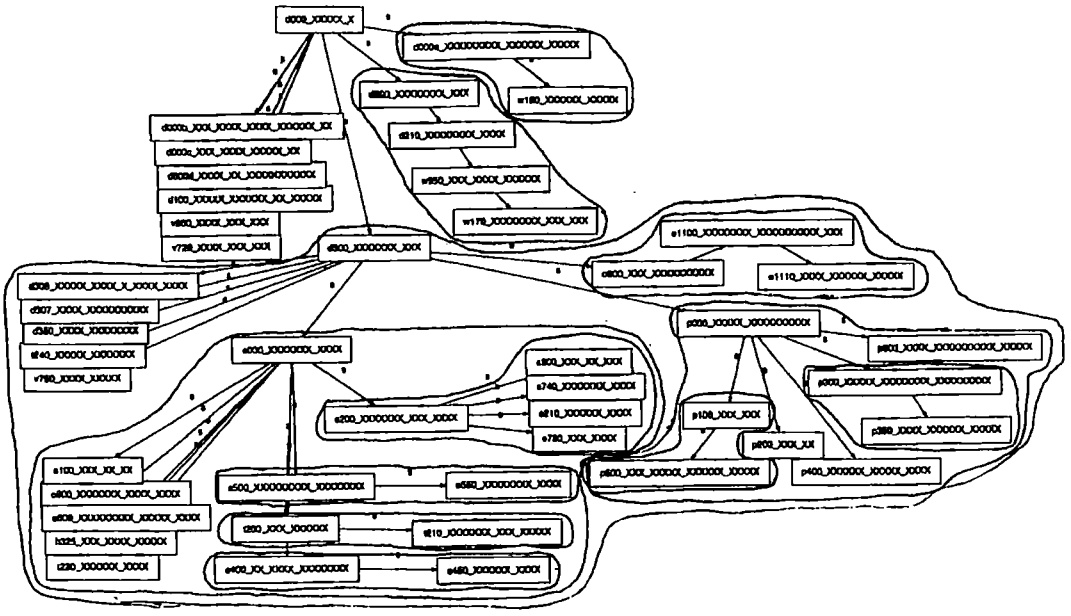


Figure 6.14: Subsets of potential reuse candidates for case study B

This process has been repeated within each of the subtrees that have been identified from the dominance tree. The initial subtree would provide the largest candidate reuse unit with respectively smaller candidates formed as constituents of the larger. However, comparing candidate reuse units across the tree, is not simply a matter of comparing the number of nodes within a subtree to estimate its size. This is due to the SECTIONs not being of equal size (lines of code). Thus, to give an indication of the number of lines of code the candidates represent, comparisons are made between the reuse candidate units' overall size. The findings indicated that, from the total set of reuse candidate units identified:

- only 10% were under 200 lines of code in length
- 50% consisted of 200-400 lines of code in length
- 25% consisted of more than 1000 lines of code in length
- 15% were greater than 3000 lines of code in length

The largest reuse candidate unit identified was around 4100 lines of code.

6.4. Step 4. Identify Data Dependencies within the Source Code

- The objective of this step is to identify which data items are accessed within the source code.

a) Identify which data items are referenced within the code

For each of the candidate reuse units identified in the previous step, the data items which are referenced are recorded, along with a note of their usage (i.e. whether the data item is created, updated, read or deleted). By categorising data items using SSADM's CRUD notation, (Creation, Read, Updated and Deleted) unnecessary interactions due to candidate reuse unit's data accesses can be identified.

A small example is now selected from case study B. The candidate reuse unit selected from this case study is highlighted in Figure 6.15.

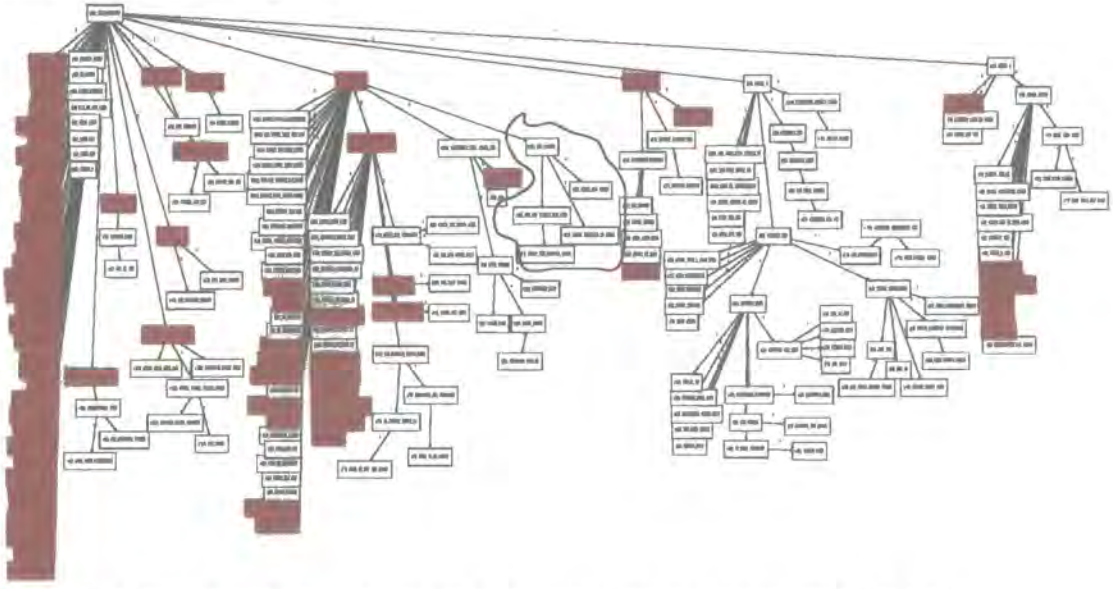


Figure 6.15: The selected subgraph from the case study B dominance tree (from Fig 6.9)

This candidate has the root node H900 and is shown in isolation in Figure 6.16.

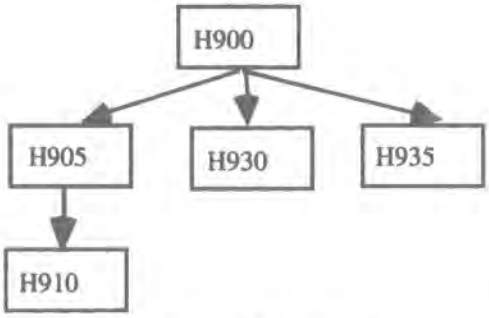


Figure 6.16: H900 reuse candidate unit taken from case study B

Table 6.1 shows the total number of data items under each category for the H900 candidates reuse unit. The IDMS database statements have not been included in the analysis as, at the present time, the analysis is restricted to COBOL source code. Little use is therefore seen, as one might expect, of

creation and deletion of data items in Table 6.1. However, the results have been included here for completeness.

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

Table 6.1: H900 reuse candidate unit

As a precursor to Step 5, there is a need to establish the data dependencies between COBOL defined GROUPs and their ELEMENTARY data items. A reduction in the number of data interactions between the candidate reuse units can provide the long-term benefits for future maintenance. Thus, through this process it is aimed to decrease the complexity of the reuse candidate units' interface. This step initiates the process where information is collected which is used to gain an indication of the costs, and benefits gained, for the restructuring.

b) Identify sets of GROUP items

By analysing the GROUP definitions in the WORKING STORAGE SECTION the hierarchical nature of the data items can be investigated. Within the case studies, large sets of GROUP definitions were found. For instance, with case study C, 102 individual groups were identified with some of the GROUPs having 6 levels of ELEMENTARY ITEMs defined.

c) Express the relationships between the members of the GROUP items

Figure 6.17 represents the groups of data items for reuse candidate unit (H900) from case study B. Data item names have been replaced with single characters to protect confidentiality. So for example, Y consists of z and B, z consists of A, and B consists of C and O.

The data items (the nodes on the graph) which are referenced within the code of the candidate reuse unit are shaded. Data items external to the candidate reuse unit are not shaded. In some cases, group items are not contained within the candidate reuse unit but are included to complete the hierarchy. Furthermore, sometimes the elementary items are not included within the candidate reuse unit but are included in the consists relation graph as this information is required for the analysis of the data interrelationships between candidate reuse units.

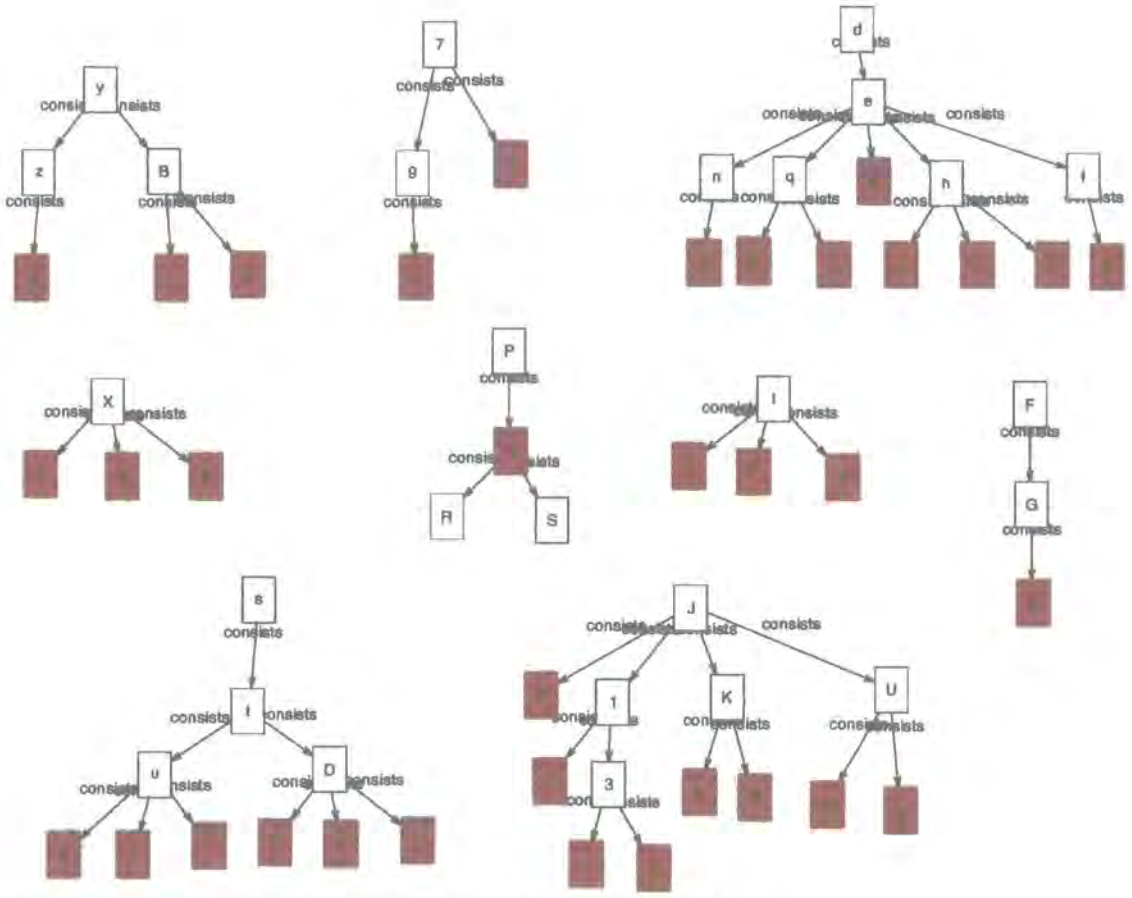


Figure 6.17: The consists data grouping hierarchy for case study B.

From visual inspection of Figure 6.17 it can be observed that there are no data dependencies between data items within the candidate reuse unit. The existence of the data dependencies can easily be identified from the graphical representation through the existence of two coloured nodes in one path. Therefore, if every possible path through the data grouping hierarchy contains only one shaded node, there are no grouping data dependencies within the candidate reuse unit.

The existence of potential data interrelationships between candidate reuse units that are not necessarily identifiable though data name comparisons, can be visually identified within the graphical representation. These data items are those nodes whose group item is shaded. Only one instance of a shaded group item was identified in the candidate module (the subtree with the group item node labelled 'Q' in Figure 6.17). The analysis of this information to investigate the actual data interrelationships is carried out by Step 5.

When analysing the interaction of data items between the reuse candidate unit and the remainder of the code, it is necessary to consider two such relationships. It is important to record how the remainder of the code affects the reuse candidate unit and also record how the reuse candidate unit affects the remainder of the code. From the analysis of Figure 6.17 it can be concluded that the reuse candidate unit is potentially affected 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, only two interacting data items have been identified and so must be recorded. These are the grandchildren on *P*, nodes marked *R* and *S*. No write operations are carried out on the supertypes. However, all write operations within the reuse candidate unit will affect the remainder of the code.

A further activity within this step involves the identification of overlapping data values. Overlapping values are evident where use is made of enumerated typing. In the H900 example from case study B, 4 sets of overlapping values have been identified, but only 1 set is involved within the H900 candidate reuse unit. In this case a data item's value overlaps with three of the four ELEMENTARY ITEMs of the GROUP. The result of the data analysis has found that there is use made of some of the data values within the reuse candidate unit consists graph as well as the remainder of the code.

Further data interrelationships also exist through the use of the redefines relationship. Within the code from which the reuse candidate unit was identified, 18 REDEFINES relationships have been located.

6.5. Step 5. Identify Data Inter-relationships Between Subtrees

- The objective of this step is to identify subtrees which use independent data items.

a) Identify areas of data independence

Areas of data independence are parts of the code which operate on totally separate subsets of the data to the remainder of the code. The results of the case studies have shown that in each of the code samples no data independent parts existed. This is to be expected with COBOL as all data items are global and therefore this reduces the likelihood of identifying data independent parts.

b) Use cluster analysis to identify the simplest interfaces between candidate reuse units

The results of the application of the clustering approaches to the case study showed that far more acceptable results were obtained from an assisted approach to clustering. The non-assisted approach's results suffer from the high use of global data within the COBOL language. Thus, the results obtained within this step are identified with an assisted approach, i.e. using the results obtained from the dominance tree.

Using Jaccard's coefficient, data interrelationships are expressed as either matches or independent data items. An independent data item is one which is only referenced within the candidate reuse unit and not within any other part of the code from which the candidate was extracted. The simplest analysis carried out involves the comparison of a candidate reuse unit's referenced data items with a specific data item name. In all cases, comparisons have been made between the candidate reuse unit and the remainder of the code from where it was extracted. While comparing candidate reuse units against one another may

yield more positive results (fewer interrelationships may exist between other single candidate reuse units), comparing the remainder of the code gives a worse case scenario.

In Figure 6.14 a number of different sized candidate reuse units are shown. These are obtained from the D000 candidate reuse unit of case study B. Two small candidate reuse units are now discussed by way of example. These are E000 and P000.

In figures 6.18 and 6.19 the E000 and P000 candidate reuse units of the D000 candidate reuse unit from case study B are shown. These were selected due to their size (each is under 2000 lines of code) rather than for having special attributes. The red nodes represent SECTIONs, while the blue and white nodes represent data items. This is only a subset of the data items, as for instance, working storage data items are excluded. The white data items represent those that are not referenced outside the specific candidate reuse unit (unique).

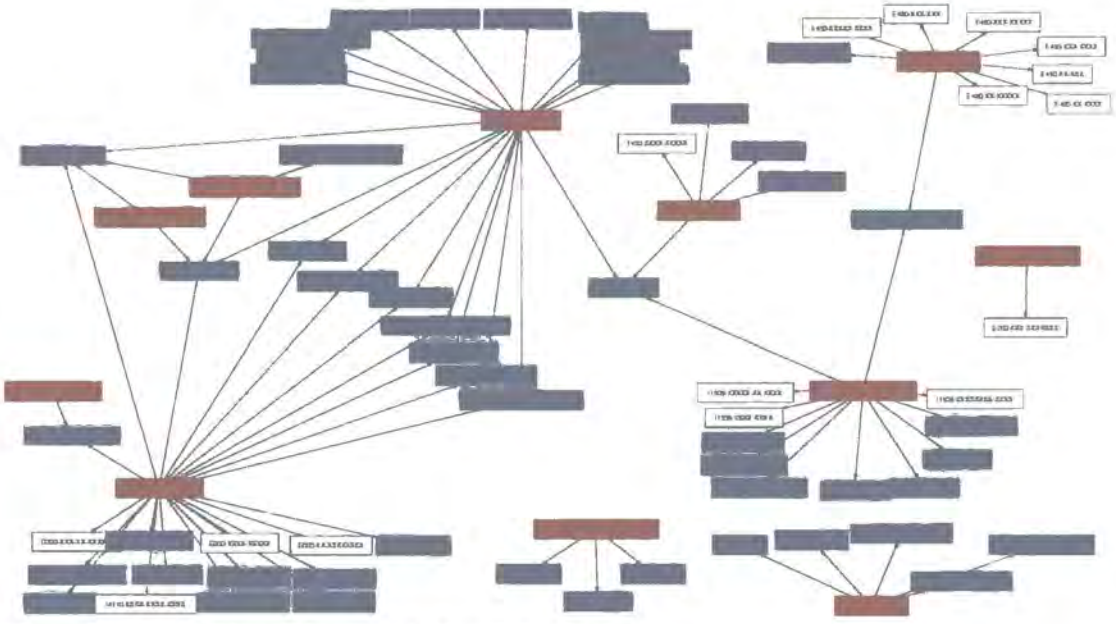


Figure 6.18: E000 subtree of case study B including data items

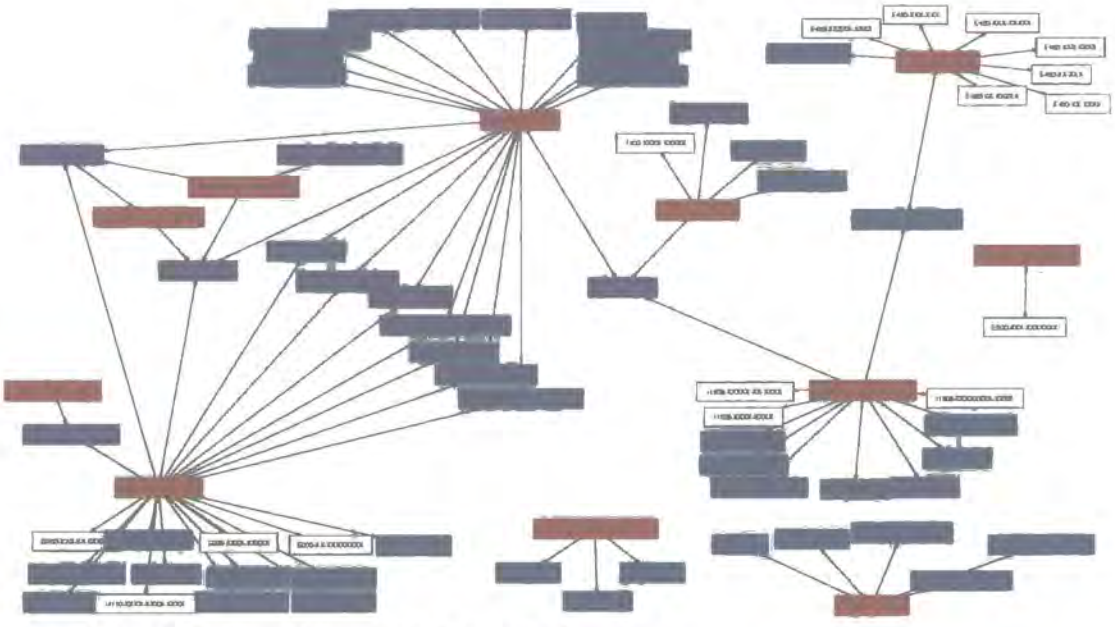


Figure 6.19: P000 subtree of case study B including data items

Obviously, for the larger reuse candidate (D000), graphical representation is infeasible as the clarity is reduced due to the complexity of the data and PERFORM interactions. In this case, D000 is a top level reuse candidate, and, therefore, the interface for this component must be compared against the remainder of the code. This relationship is represented graphically within Figure 6.20.

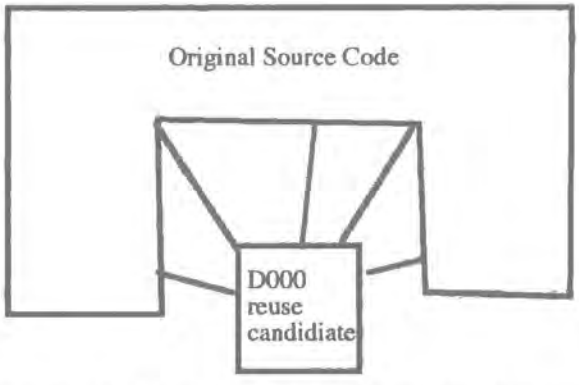


Figure 6.20: Identifying data interrelationships between reuse candidates

Due to the complexity of the interfaces between components instead of a graphical representation, a table is used. This describes the data interactions for the D000 candidate reuse unit compared with the remainder of the code. The results are given in Table 6.2 below.

Total number of data items in code	1502
Total number of data items in candidate reuse unit	602
Data items unique to candidate reuse unit	160
Data items not unique to candidate reuse unit (the candidate interface)	442

Table 6.2: Data usage for D000 candidate reuse unit

Overall, our results indicate that only 10% of the data items were independent of all the code. From Table 6.2 it can be seen that 73% of the data items within the candidate reuse unit could potentially be read or updated by the remainder of the COBOL code. In terms of gaining an encapsulated reuse candidate, these data interrelationships need to be recorded, as the relationships will, after encapsulation, result in the necessity to pass data values between candidate reuse units. As far as possible the high degree of interactions needs to be reduced.

From analysing the relationships of each of the individual candidate reuse units to the remainder of the code, it is then possible to assess the interfaces between groups of candidates reuse units. Taking the case of the two reuse candidates (P000 and E000) which have been shown above in Figures 6.18 and 6.19, the interface between them is shown in Figure 6.20.

E000 consists of 1607 lines of code and 227 data items. P000 consists of 654 lines of code and 58 data items. There are a total of 257 data items within the two candidate reuse units. 28 data items form the interface between the two candidate reuse units. These are shown in Figure 6.20. Thus 10.9% of the data items used within the candidates are dependent on the other candidate reuse unit; 48.3% of P000 and 12.3% of E000.

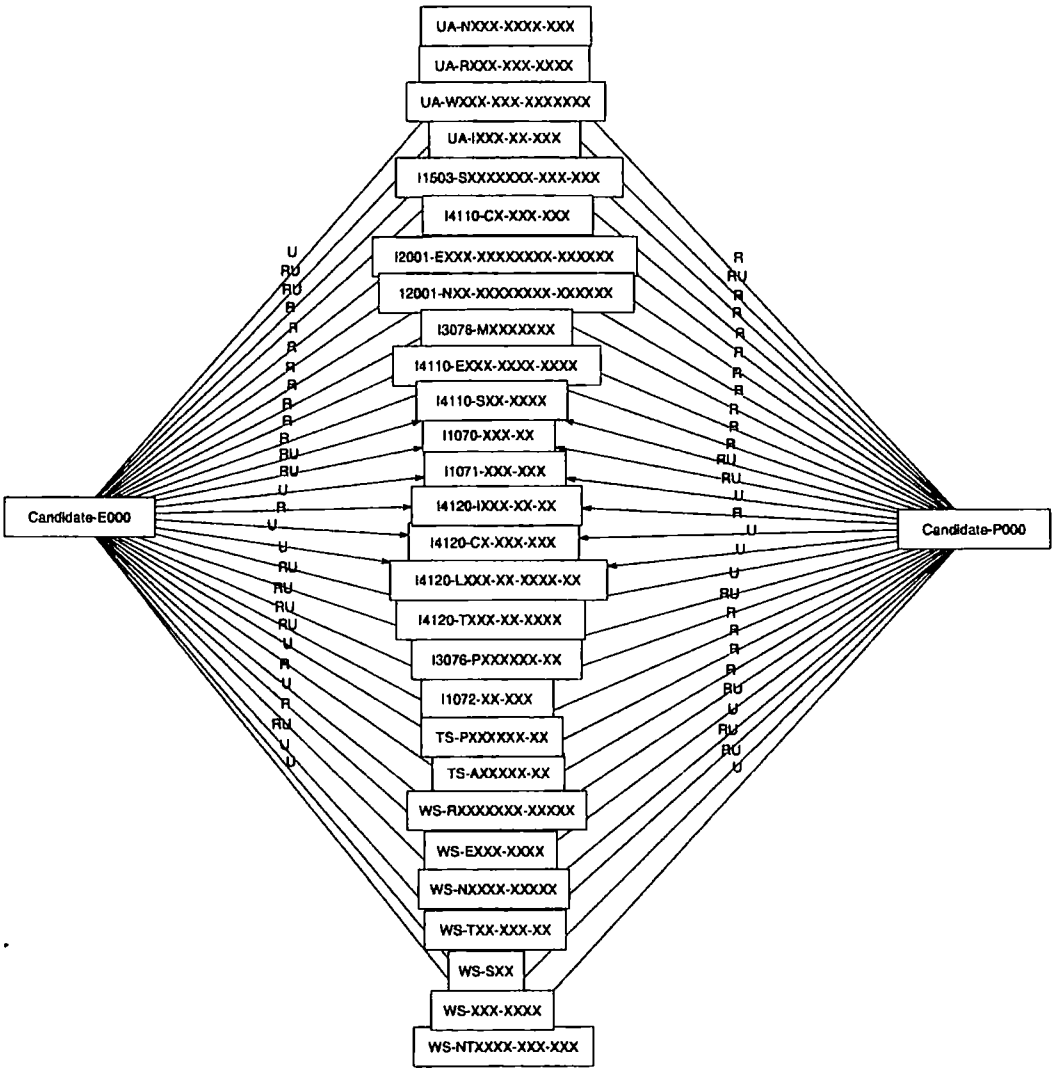


Figure 6.21: The interface between the P000 and E000 reuse candidates

At first we assume that a two way relationship exists between each of the data items within the interface. Thus, both candidates reuse units read and update each of the data items and consequently there is a need to pass the data items to and from each of the candidates. However, in many cases this two way relationship does not exist and therefore the amount of message passing can be reduced.

To further investigate the interface between the two candidate reuse units it is necessary to analyse the interrelationships between the data items. For accurate results, the definition of a candidate reuse unit's interface would normally be considered by comparing it to the remainder of the code. In this way, the interfaces between the other candidates are indirectly derived. However, by way of example, we analyse the two candidate reuse units in isolation and assume that in total they constitute the entire COBOL module.

An interface investigation was carried out using the SSADM notations. A letter was allocated for the usage of the data for each data item. R was assigned to data items which had been Read, and U for data items which had been updated. The analysis process also collects Initialisations and Deletions of data

items. However, no examples of these data usages were found in the P000 and E000 candidate reuse units. Table 6.3 summarises the results of the analysis.

	E000			P000			
Data Item	R	U	Total	R	U	Total	Usage
1. I1070-XXX-XX	2	1	3	7	4	11	RU,RU
2. I1071-XXX-XXX	3	3	6	16	8	24	RU,RU
3. I1072-XX-XXX	2	2	4	9	9	18	RU,RU
4. I2001-EXXX-XXXXXXX-XXXXXX	4		4	4		4	R,R
5. I2001-NXX-XXXXXXX-XXXXXX	5		5	7		7	R,R
6. I3076-MXXXXXXX	1		1	16		16	R,R
7. I3076-PXXXXXX-XX	8	2	10		2	2	RU,U
8. I4110-EXXX-XXXX-XXXX	1		1	1		1	R,R
9. I4110-SXX-XXXX	1		1	2		2	R,R
10. I4120-CX-XXX-XXX	1		1	1		1	R,R
11. I4120-IXXX-XX-XX		4	4		4	4	U,U
12. I4120-LXXX-XX-XXXX-XX		4	4		2	2	U,U
13. I4120-TXXX-XX-XXXX		4	4		2	2	U,U
14. TS-AXXXXX-XX	34	1	35	1		1	RU,R
15. TS-PXXXXXX-XX	6	1	7	5		5	RU,R
16. UA-NXXXXX-XXXX-XXX		2	2	12		12	U,R
17. UA-IXXX-XX-XXX	3		3	1		1	R,R
18. UA-RXXX-XXX-XXXX	7	4	11	7	1	8	RU,R
19. UA-WXXX-XXX-XXXXXXX	4	3	7	1		1	RU,R
20. WS-EXXX-XXXX	1		1	2		2	R,R
21. WS-NXXXXX-XXXXX		4	4	3	65	68	U,RU
22. WS-RXXXXXXX-XXXXX		1	1	1		1	U,R
23. WS-SXX	1	119	120	1	13	14	RU,RU
24. WS-TXX-XXX-XX	2		2		1	1	R,U
25. I1503-SXXXXXXX-XXX-XXX	1		1	1		1	R,R
26. I4110-CX-XXX-XXX	1		1	1		1	R,R
27. WS-NTXXXXX-XXX-XXXX		4	4		41	41	U,U
28. WS-STXX-XXXX		1	1	1	2	3	U,RU

Table 6.3: Data item usages within the P000 and E000 candidate reuse units

Using Table 6.3 it is possible to summarise the data item usage between the interface of the two candidate reuse units. This is achieved by looking at the usage of each data item and classifying them accordingly.

Each of the relations refer, with a fully parameterised interface, to the input and output of the data flow within the COBOL module. Table 6.4 is a look-up table for assessing necessary reuse candidate unit input and output parameters. Thus, for the E000 and P000 candidate reuse units, the usages from Table 6.3 are used to define the necessary set of input and output parameters between them.

Usage	Input (E000)	Output (E000)	Input (P000)	Output (P000)
(RU,RU)	X	X	X	X
(R,R)	X		X	
(U,U)		X		X
(U,R)		X	X	
(R,U)	X			X
(U,RU)		X	X	X
(RU,U)	X	X		X
(RU,R)	X	X	X	
(R,RU)	X		X	X

Table 6.4: Usage translation to parameterisation of the interface

So for example, from Table 6.3 data item 1 (I1070-XXX-XX) usage is identified as RU,RU and therefore from Table 6.4 it can be identified that I1070-XXX-XX should be input and output to both E000 and P000. However, with data item 7 (I3076-PXXXXXX-XX) its usage is defined as RU,U and thus it is unnecessary to provide input to P000.

The information available from this step can be used to begin a process of reducing the data items that are presently considered. This process is carried out in detail in Step 9.

Above, only the simple naming interrelationships are considered. Item inter-dependencies based on the information gained from Step 4 have also been investigated. It was found that for the COBOL code analysed, the grouping interrelationships were not a problem. Only 4% of data items by grouping were involved in interactions. In general, it was found that, where grouping had been used, the identification procedures usually grouped all related elementary items into one candidate reuse unit.

6.6. Step 6. Identify Potential Reuse Candidates from Users / Designers of the Code

- The objective of Step 6 is to apply a top-down approach to the identification of reuse candidates with the assistance of the users and designers of the code.

a) **Use experts to describe the functionality of the candidates proposed.**

Interpretation of the candidate reuse unit's functionality, hence the process of concept assignment was found to require a great deal of domain knowledge. Interpretation by non-experts could be made only at a low level, for example, the more general functionality such as the manipulation of dates. However, from interviewing the experts to investigate their views on possible functions of the program, three main areas were identified. These are:

- Interaction with the users
- Validation of input
- Updating the database

With further investigation with the assistance of the experts, it was possible to identify the relationships between the low level components identified automatically from the source code (Figure 6.22).

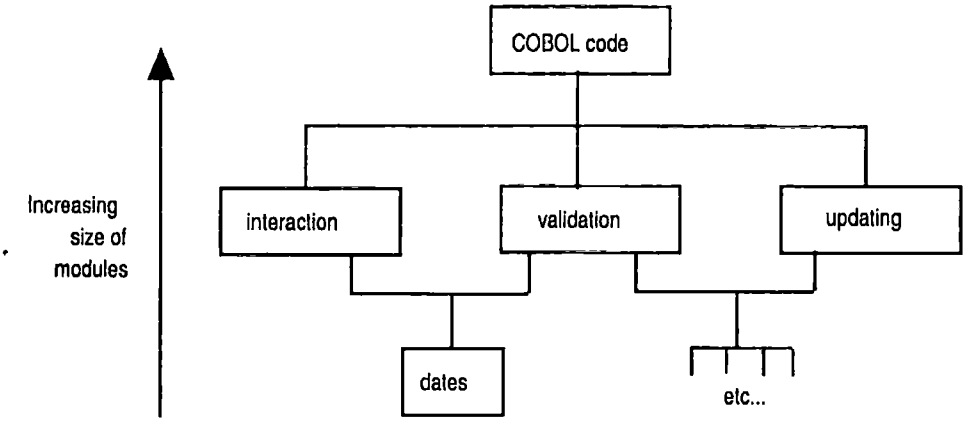


Figure 6.22: Hierarchy of components identified.

Thus, a number of reuse candidates were selected from the set of candidate reuse units identified from the previous steps.

b) **Encourage experts to discuss improvements to candidates identified.**

The experts indicated that there was a high degree of interaction between the SECTIONs providing the above list of functions (i.e. interaction, validation and updating). Given this information, it is possible to tailor the simplification procedures to focus on the areas of provision of code implementing these functions. Furthermore, given SECTIONs with high degrees of data interactions the experts were encouraged to identify SECTIONs that would be better included within other reuse candidates by way of reducing the overall data interactions. The results of the case studies showed that experts were able to gain a good understanding of the composition of the reuse candidates by simply using the SECTION names as indicators.

The involvement of users and designers is not exclusive to this step. Their expertise is used throughout the method, for instance, with the concept assignment process and the reuse candidate cost / benefits analysis process. This step, however, is the point when formal involvement of the users and designers is sought as this step will account for the greatest use of their time.

6.7. Step 7. Identify Potential Simplification Procedures to Assist Encapsulation

- The objective of this step is to investigate heuristics for the simplification of the PERFORM structure and data interactions of the graph to reduce the complexity of its interactions and therefore assist the encapsulation process.

SECTION Relationships

a) Locate non-functional requirements from those typical within the domain

Since this is the first application of the method it was not possible to draw upon past experience in order to identify typical non-functional requirements within the domain. For this reason, the list was composed of only the application specific requirements.

b) Compose an application specific set of non-functional requirements

A number of application specific requirements were identified. However, many of the more general ones cannot be identified here due to confidentiality agreements. Therefore, examples are given within this step of those non-functional requirements that were identified within the case studies but which are more general in nature. For instance, the requirement of error handling was found within each of the three code samples analysed.

c) Identify actual SECTIONs within the code that implement the non-functional requirements

The complexity of the PERFORM graph can be reduced by the temporary exclusion of non-functional aspects of the code.

The results of the analysis, followed by the removal of the error-handling routines can be seen in Figure 6.23b from case study B compared with the original graph shown in Figure 6.23a. (This is the same figure as shown in Figure 6.2 but with the SECTION names truncated to 4 characters to reduce the width of the graph.) Within this code sample, 10 SECTIONs which deal solely with error handling were identified. One of the SECTIONs was found to be called by approximately 90% of the other

SECTIONs and, upon analysis of the code, was found to be concerned with error-handling for the database routines. A further 9 SECTIONs were identified by the document analysis tasks (Task b) and were mainly concerned with the display of error messages. Although a greater number of SECTIONs were identified using the document analysis, the resulting simplification of the PERFORM graph was far less marked due to the low number of PERFORM linking these SECTIONs. The analysis showed that neither of the tasks alone were able to identify all of the error handling occurrences. For instance,

- SECTIONs concerned only with error conditions
    - by identifying those named ERROR, ERR etc.
    - by identifying those whose descriptions describe the process of error handling
- data items not concerned with the functionality of the program
    - data items which only carry key codes for error messages

d) Analyse area of high fan-in within code

The result of removing the non-functional aspect of error handling within Figure 6.23b shows a significant reduction in the number of arcs connecting the remaining nodes. In this example, over 200 PERFORMs have been removed. The removal of these arcs reduces the direct dominance nodes and so allows the identification of larger candidate reuse objects. However, from observations of the graph, it is possible to see other areas of densely grouped arcs. These may represent further non-functional aspects of the code and, therefore, the above procedures should be repeated until no further non-functional aspects can be identified.

e) Remove identified SECTIONs from the code

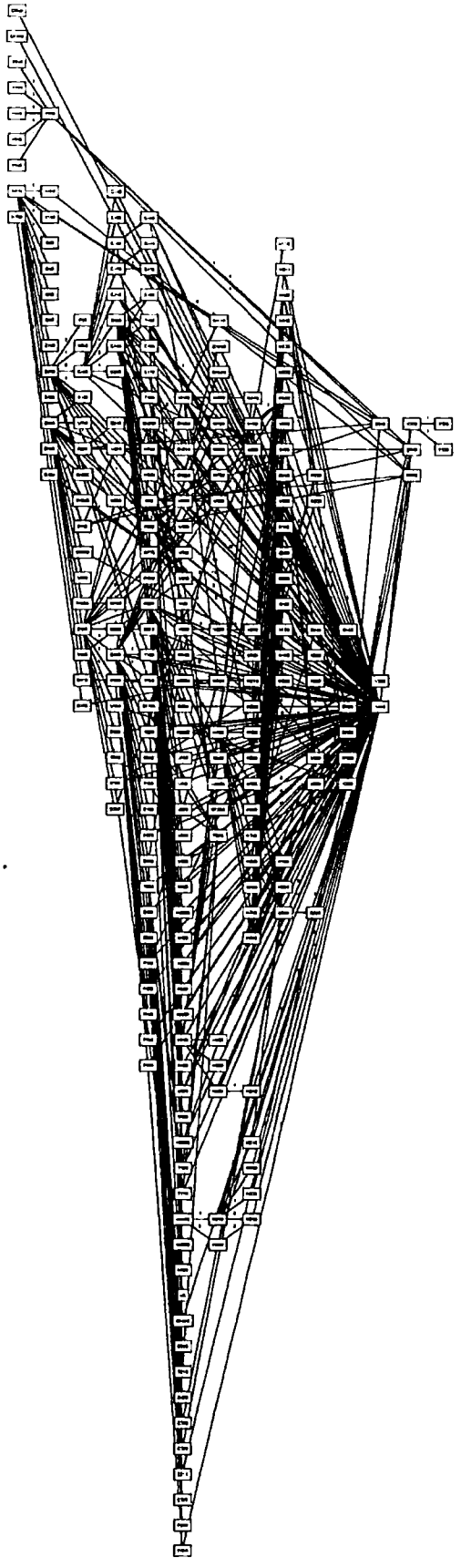


Figure 6.23a: Initial call graph

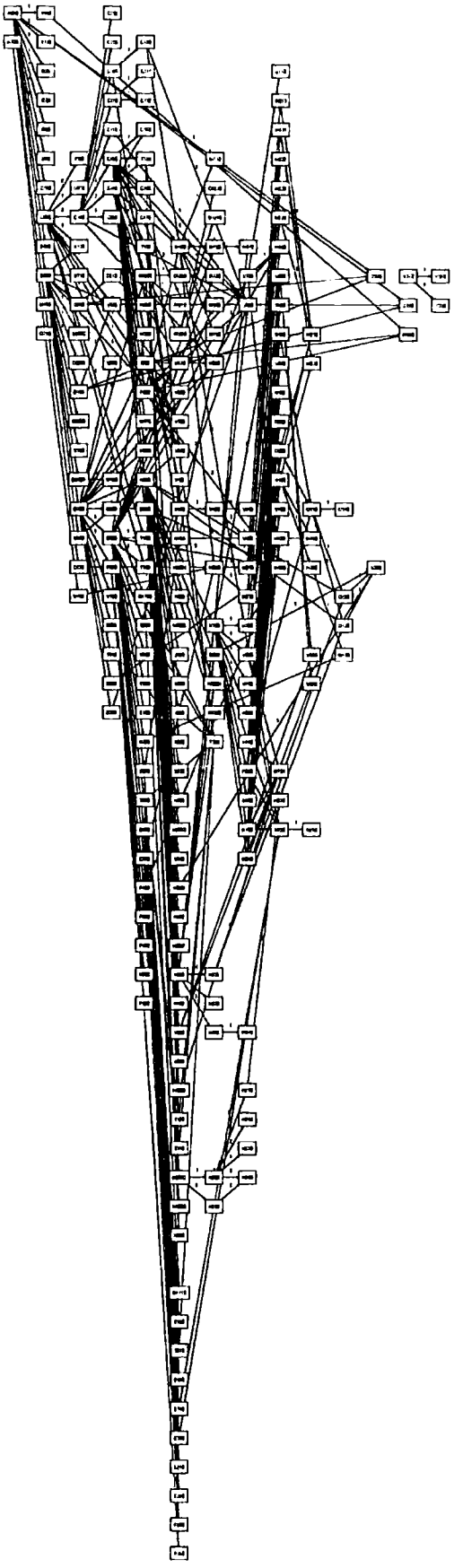


Figure 6.23b: Call graph upon removal of error routines

In the examples above, it was possible to identify error handling functions, and therefore examples of the implementation of non-functional requirements, at least partly through the use of fan-in analysis. Furthermore, it was found that, in those cases where the requirement has been implemented, the non-functional requirements were always in the form of a dedicated SECTION. The advantage of this approach is that, as all the error-handlers are implemented together, it is likely that a consistent approach will be found throughout. Our analysis failed to highlight any cases to contradict this hypothesis.

Data Relationships

Within the case study, other simplification procedures have also been applied and, therefore, it includes the removal of a number of data relationships using many different strategies. These are described with the results obtained from their application within the step's task headings below.

a) Identify data items not implementing specific functionality

Data items that can be removed from data interface assessment without affecting the operation of the program is a good initial form of simplification. Within the case studies, those data items listed in LINKAGE SECTION were removed. The removal of data items not describing the functionality of the program resulted in a small but not insignificant reduction within the overall graph. For instance, the removal of error handling data items within the case study B resulted in a reduction of 6% of the data items.

b) Identify data items which are not updated throughout the code

The removal of data items that are never updated in the life-time of the program's operation also assist the simplification process. Within the case studies, those data items that are only read during the program were removed. The removal of these data items made a significant contribution to increasing the simplicity of the graph, for instance, the removal of those data items representing reference points within the program i.e. look-up-codes, resulted in a reduction of 24% of the data items within the initial graph.

c) Identify data items already external to the program

The removal of the data items which are external to the program also aid the simplification of the data interactions. In this case, these data items cannot be local since they are used externally to the program. These can therefore be removed from the investigation to identify local data. The removal of data items from the LINKAGE SECTION and therefore those already external to the program within case study B resulted in a reduction of 13% of the total data items.

Furthermore, from consulting the experts (Step 6), it was possible to identify many data items that could be removed through our simplification procedures. By applying some of the procedures that the experts identified, a reduction of nearly 50% was achieved in the number of data items requiring investigation. 43% of the data items removed from investigation resulted from the use of three main strategies described above.

6.8. Step 8. Select Subtree(s) to Form Reuse Candidates using Graph Slicing

- The objective of this step is to select SECTIONs of the code which show they have potential for reuse.

Step 8 is the first step towards beginning the development of the selected reuse candidates. This step involves slicing the parts of the code identified by previous steps into common groups of statements suitable for reuse.

a) Collate information on the suitability of candidates and assess their viability

Within case study A, no reuse candidates have been identified for reengineering. Although 2 potential reuse candidates were identified (the subgraphs of A100 and A300), these were eventually rejected as it was felt that the benefit to cost ratio was too high. For instance, the largest of these reuse candidates (A300) was under 100 lines of code. However, this was more or less to be expected due to the small size of the original code module.

In case study B some larger reuse candidates were identified, one was over 4000 lines of code. However, out of the twenty reuse candidates identified, only four of these were considered as worthy of continued analysis beyond this step. These reuse candidates can be seen in Figure 6.24. As in case study A, many of the reuse candidates were rejected within Steps 4 to 6 as it was decided that the costs of making them reusable were greater than the benefits that would be gained from their use due to their small size. In addition, more reuse candidates may be considered later when the splitting of functionality is analysed within Step 10. For instance, a typical example would be those reuse candidates which are restricted in size due to their inclusion of direct dominance nodes (the shaded nodes). Therefore, at a later stage of the method, it may be decided to increase the granularity of some of the reuse candidates, based on discussions with likely users of the reuse library.

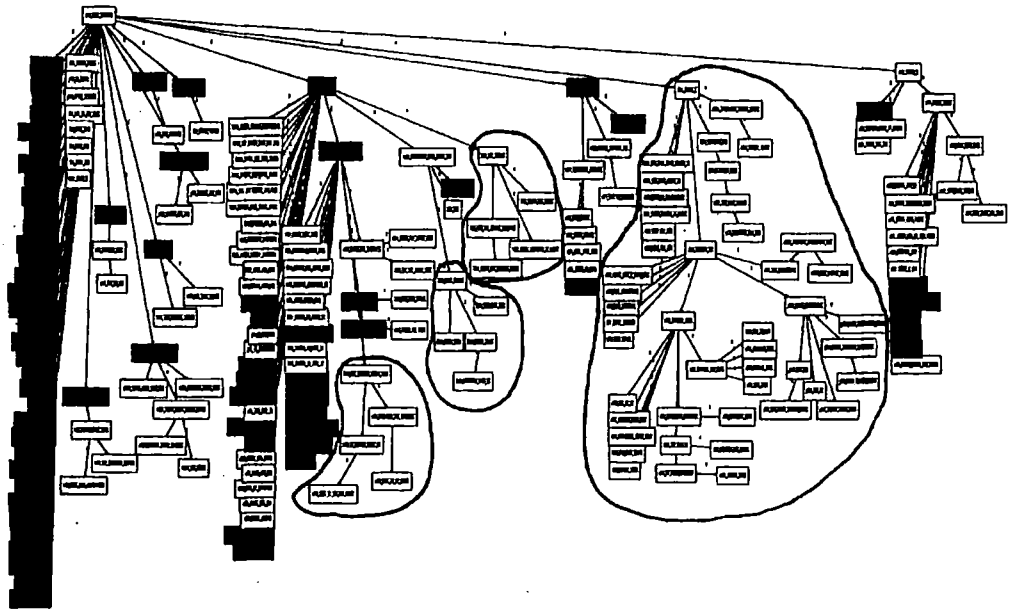


Figure 6.24: Reuse candidates identified in case study B

For case study C, seven reuse candidates were originally identified. These included reuse candidates with direct dominance nodes. At this stage insufficient analysis has been carried out to assess if the benefits of obtaining the reuse candidates are sufficient to warrant the necessary reengineering process to make these candidates actually reusable. Therefore, at this stage the process is restricted to those reuse candidates without direct dominance relationships. Thus, as can be seen in Figure 6.25, a total of 7 reuse candidates are available (although at considerably lower levels of granularity to those identified in the earlier stages).

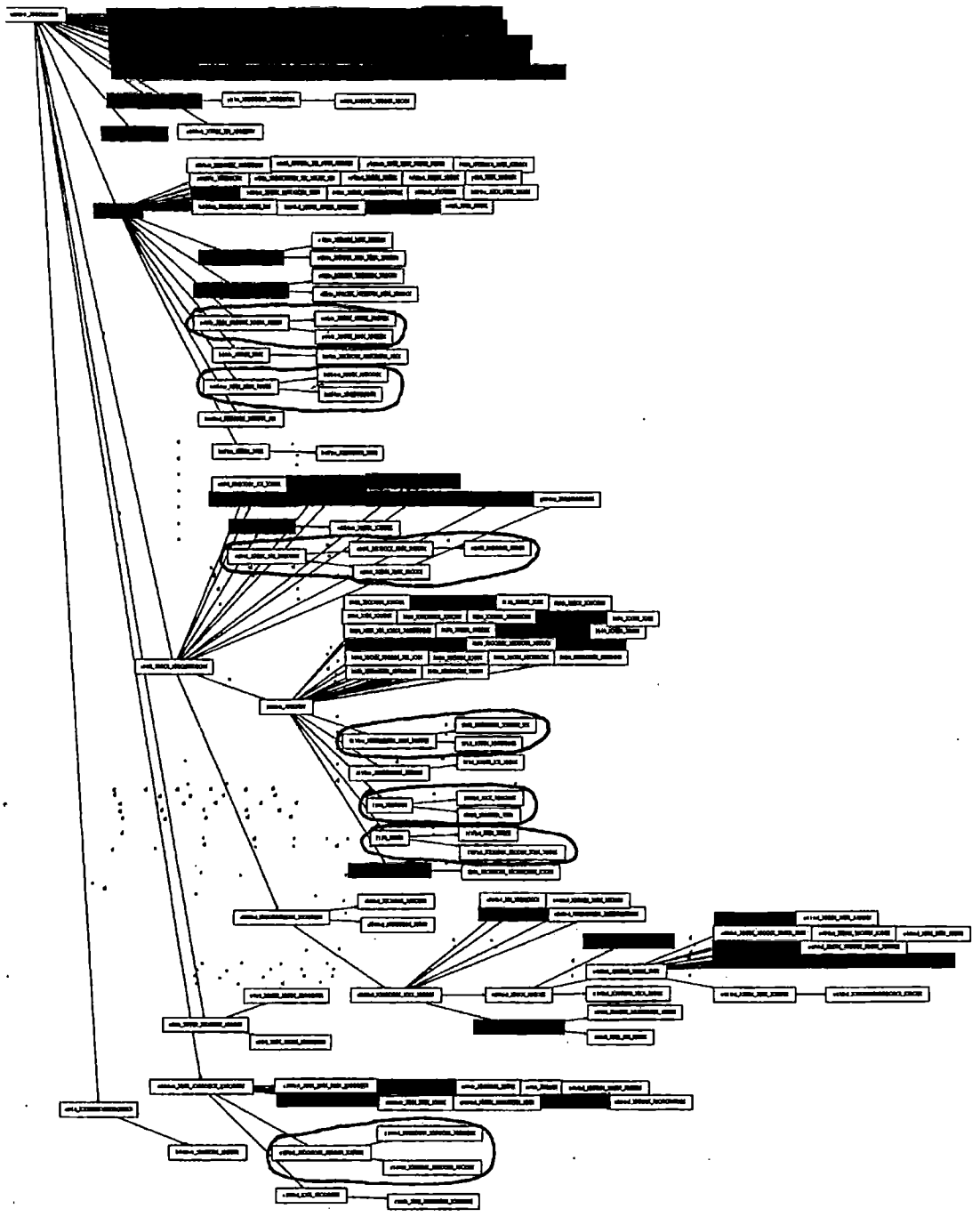


Figure 6.25: Reuse candidates identified in case study C

The candidates within case study C are far smaller that in case study B (around 500-1000 lines of code as opposed to up to 4000 lines of code). In this case, unlike case study A, it is not the size of the COBOL module that results in small reuse candidates. Case study C is the largest of the code modules. In this case reuse candidates are restricted in size due to the complexity of the PERFORM structure.

In general, it has been found that the overall reuse / reengineering process often favoured the larger of the reuse components. Therefore, the large reduction in reuse candidates considered (especially in case

study B) was because many of them could be grouped into larger reuse candidates (the granularity issue).

b) Show exact composition of each candidate using graph slicing.

Finally in this step, the first actions are taken to show the exact composition of the proposed reuse candidates. This is carried out with the graphical representations and slicing at the SECTION level. A sample of reuse candidates from case study B is shown in Figure 6.26 from 6.12.

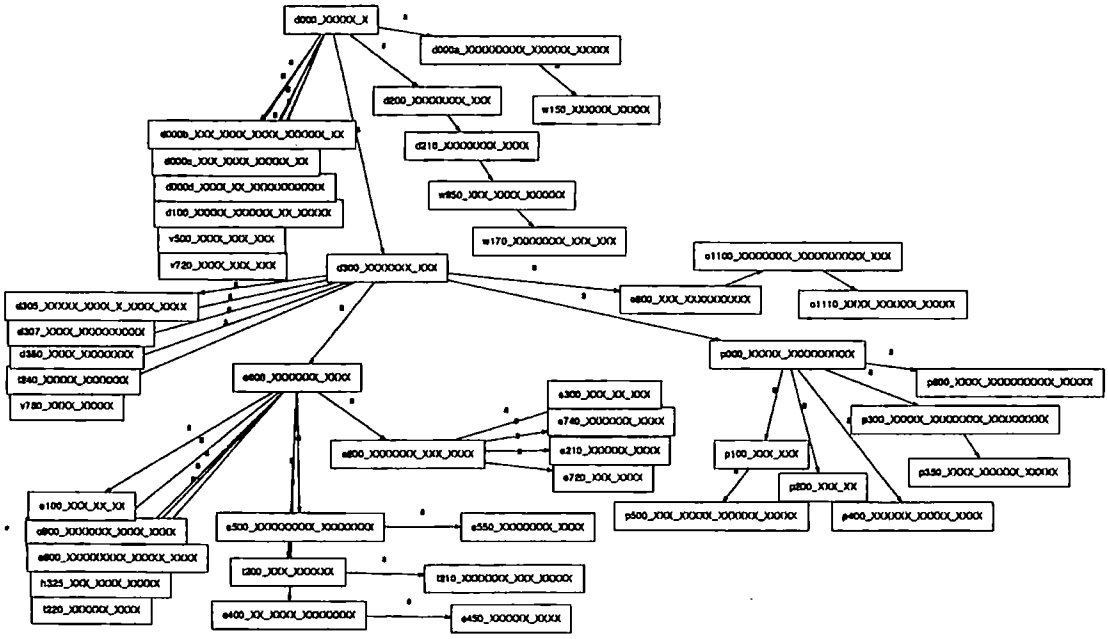


Figure 6.26: Example reuse candidate sliced from the code of case study B

Within the case study it was found that the graphical representations allowed discussions between those carrying out the reuse / reengineering process and potential users of the reuse candidates. In addition, the availability of the PERFORM / dominance relationships was useful for providing documentation for the reuse candidate when added to the reuse library. The graphical representations were particularly useful in assisting the concept assignment process.

6.9. Step 9. Identify Data Items in Reuse Candidates that would Reduce Data Interactions

- The objective of this step is to identify where the definition and usage of data items occurs across SECTIONs within different reuse candidates.

a) Identify types of interface for each of the reuse candidates

By sorting Table 6.3 into the categories implied by Table 6.4 of Step 5, it is possible to indicate the type of interface that should be developed for each of the data items between the two reuse candidates. The result of this analysis is shown in Table 6.5.

Candidate E000		Candidate P000		Data item
In	Out	In	Out	
R	U	R	U	1. I1070-XXX-XX 2. I1071-XXX-XXX 3. I1072-XX-XXX 18. UA-RXXX-XXX-XXXX 23. WS-SXX
R		R		4. I2001-EXXX-XXXXXXX-XXXXXX 5. I2001-NXX-XXXXXXX-XXXXXX 6. I3076-MXXXXXXX 8. I4110-EXXX-XXXX-XXXX 9. I4110-SXX-XXXX 10. I4120-CX-XXX-XXX 17. UA-IXXX-XX-XXX 20. WS-EXXX-XXXX 25. I1503-SXXXXXXX-XXX-XXX 26. I4110-CX-XXX-XXX
	U		U	11. I4120-IXXX-XX-XX 12. I4120-LXXX-XX-XXXX-XX 13. I4120-TXXX-XX-XXXX 27. WS-NTXXXXX-XXX-XXXX
	U	R		16. UA-NXXXXX-XXXX-XXX 22. WS-RXXXXXXX-XXXXX
R			U	24. WS-TXX-XXX-XX
	U	R	U	21. WS-NXXXXX-XXXXX 28. WS-STXX-XXXX
R	U		U	7. I3076-PXXXXXX-XX
R	U	R		14. TS-AXXXXX-XX 15. TS-PXXXXXX-XX 19. UA-WXXX-XXX-XXXXXXX
R		R	U	

Table 6.5: Data Usage per data item

The information available within Table 6.5 can be used to begin a process of reducing the data items that are presently considered within the reuse candidate's interface.

b) Reduce interface by removing or reducing specific sets of relations.

As indicated in Chapter 4, using the results of Step 5, when a candidate's shared data items form a (R, R) relationship, then these data items are not updated within either candidate. The reduction on the interface (original shown in Figure 6.20) between P000 and E000 is shown in Figure 6.27. If this is the case throughout the entire COBOL module, then these are more like constants and therefore in many cases it is possible to avoid the necessity of passing the values of these data items.

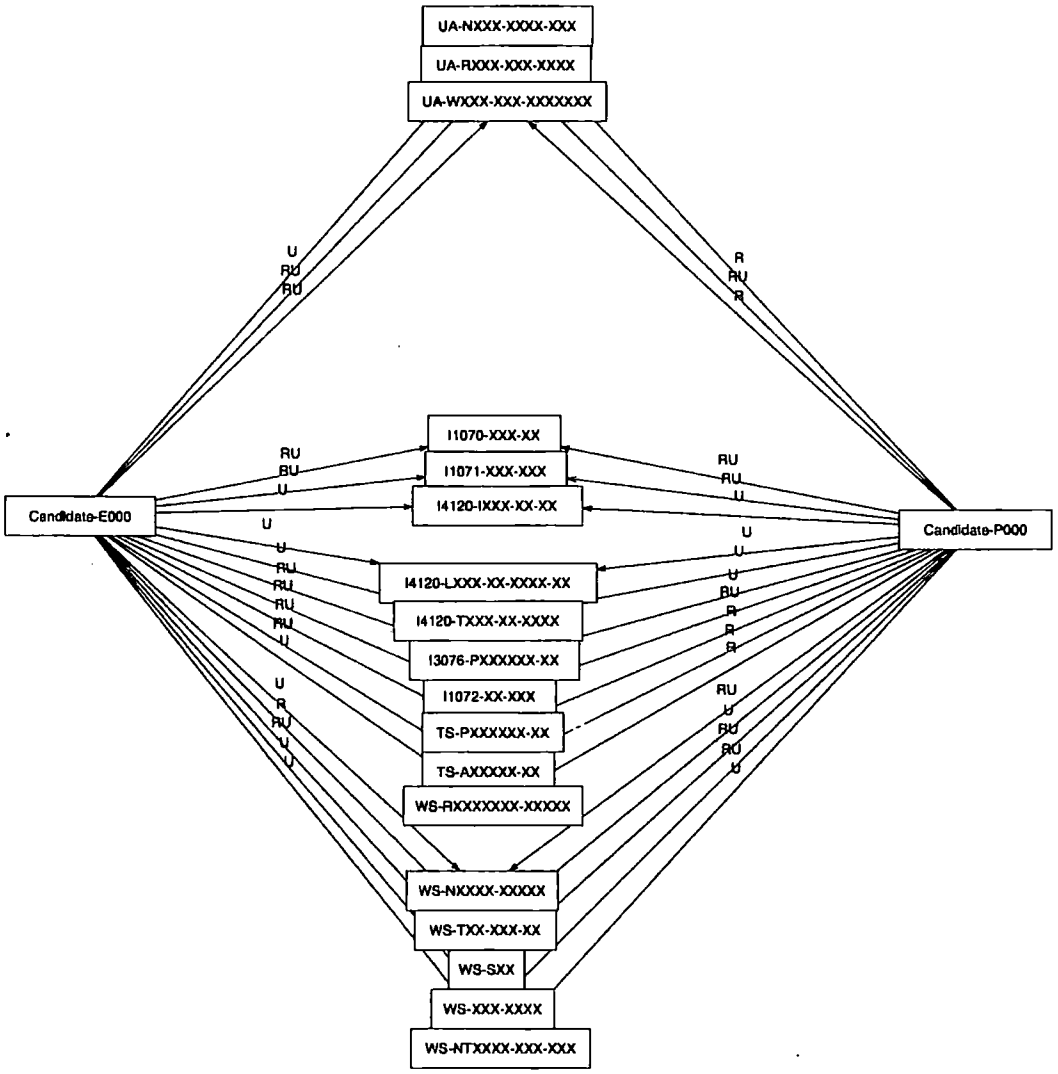


Figure 6.27: Interface of P000 and E000 with (R, R) relationships removed

Further to this process, the (U,U) operations may be instances of local variables as each module will update the data item and ignore the operations of the other reuse candidate. The removal of the (U, U) relation from the P000 and E000 interface is shown in Figure 6.28.

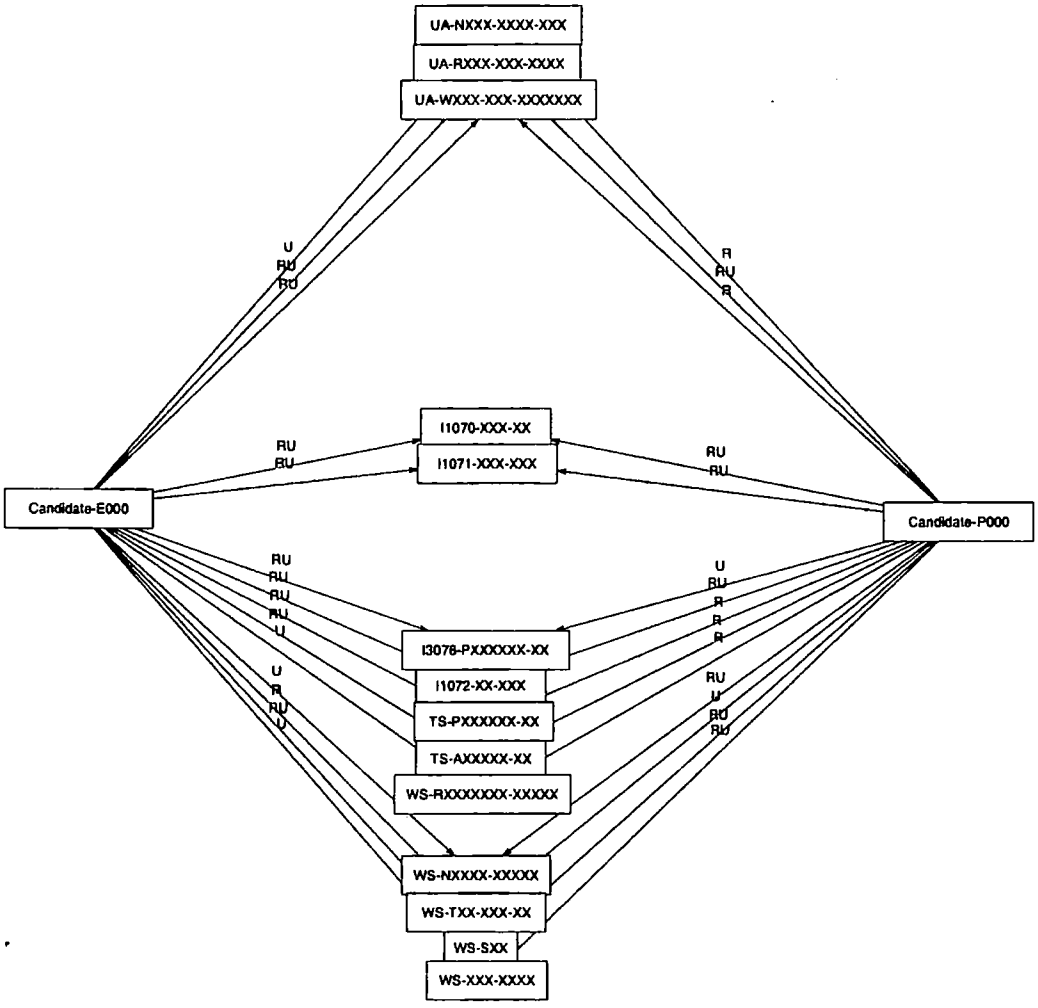


Figure 6.28: Interface with (R, R) and (U, U) relationships removed

Within the case study, the two relationships (i.e. (U, U) and (R, R)) constitute a reduction of 14 from a total of 28 data items within the interface (taken from Table 6.5).

The above examples demonstrate cases where the interactions between the two reuse candidates can be ignored. In addition, Chapter 4 indicated that it is also possible to, although not eliminating the interactions entirely, reduce the amount of formal passing of data. For instance, if a candidate only carries out an update operation on a specific data item, there is no point in the opposing reuse candidate out putting the value of the data item as it will not be read. Therefore, any single 'U' operation (i.e. (U, R), (R, U), (U, RU) (RU,U)) means that the other reuse candidate need not provide an output value. Within the case study, this affects a further 6 of the data items. Using the same approach, any single 'R' operation (i.e. (U, R), (R, U), (R, RU) (RU, R)) means that the reuse candidate need not output the data item, as the value of the data will not have changed. Within the case study, this also affects a further 6 of the data items.

Removing the necessity to input or output specific data items is a different process from the removal of (R, R) or (U, U) relationships. The reduction of input / output requires that only specific directions

of the interface are removed. Therefore, in Figure 6.29 the directions are indicated by separate relationships.

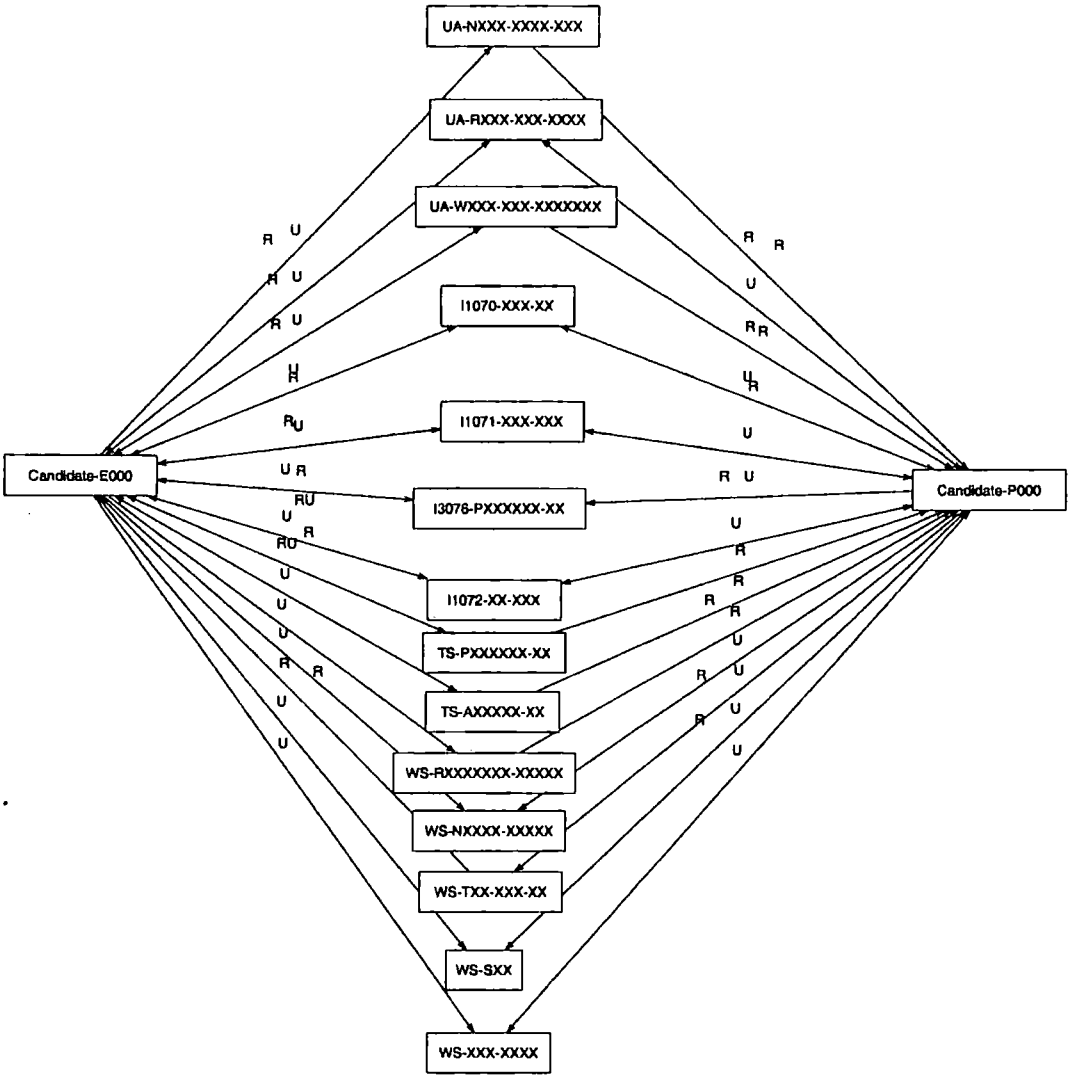


Figure 6.29: Showing input and output of the candidates' data items.

Thus, in carrying out these simple tasks, a large reduction has been demonstrated in the number of data items required to be passed between the reuse candidates. Further reductions can also be made by considering the ordering of the read / update operations. This, however, would require control flow analysis that is not considered within this thesis.

In the largest reuse candidate from case study B (D000), one strong proposal for the movement of a created, but unused, data item which could be relocated to another reuse candidate has been identified. This would have the benefit of reducing the necessity of interactions between the two reuse candidates. Furthermore, it has also been identified that a number of other possible data items exist, but which have more complex interactions within the code. Therefore, the benefit of moving these will have to be considered along with a detailed cost analysis for the reengineering process.

6.10. Step 10. Identify SECTIONs where Slicing could Assist Separation

- The objective of this step is to identify SECTIONs which implement more than one functionality.

a) Identify SECTIONs suitable for slicing.

This is an area where further work is required. However, at present some SECTIONs, that could offer more than one functionality, have been identified. Ideal candidates for splitting are those SECTIONs present within cycles that demonstrate high connectivity.

b) Carry out detailed analysis to investigate if slicing is feasible

Two approaches were described within Chapter 4 for slicing SECTIONs. These are:

- Subgraphs of data - graphs of data dependencies are generated to see if there are any disconnected subgraphs.
- Ranking - re-location of data items are proposed based on their usage within each reuse candidate.

Examples of how the analysis of the data dependency graphs and ranking process have been used within the case studies are given.

Within case study C, it was indicated that a large number of nodes were involved within a cycle. This cycle was removed before the dominance tree was generated. The disadvantage of such an approach is that the reuse candidates that are obtained are much smaller than could have otherwise been achieved. By looking at a graph of only those nodes involved within the cycle, potential SECTIONs can be identified which when split remove the cycle (i.e. those providing multiple functions). The graph of the cycle within case study C is shown in Figure 6.30.

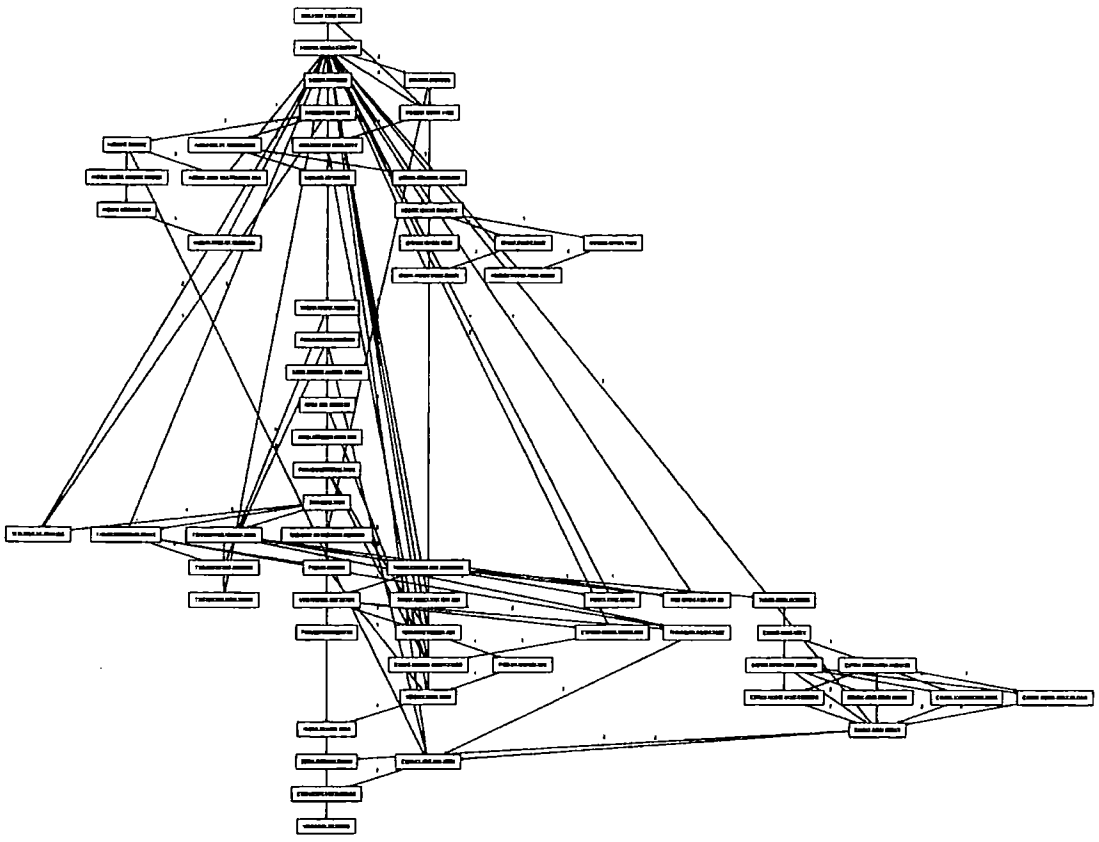


Figure 6.30: The nodes within a Cycle of Case Study C (grouped node in Figure 6.7)

In order to break the cycle, it is necessary to slice one or more SECTIONs. Two SECTIONs within this cycle are now analysed. These are called T110 and V100. The data dependency analysis of V100 identifies that its dependency graph is fully connected. This is shown in Figure 6.31. Thus, since at this stage it is likely that there will be a number of potentially more feasible SECTIONs to slice, in this case it is likely to be impractical to reengineer this SECTION.

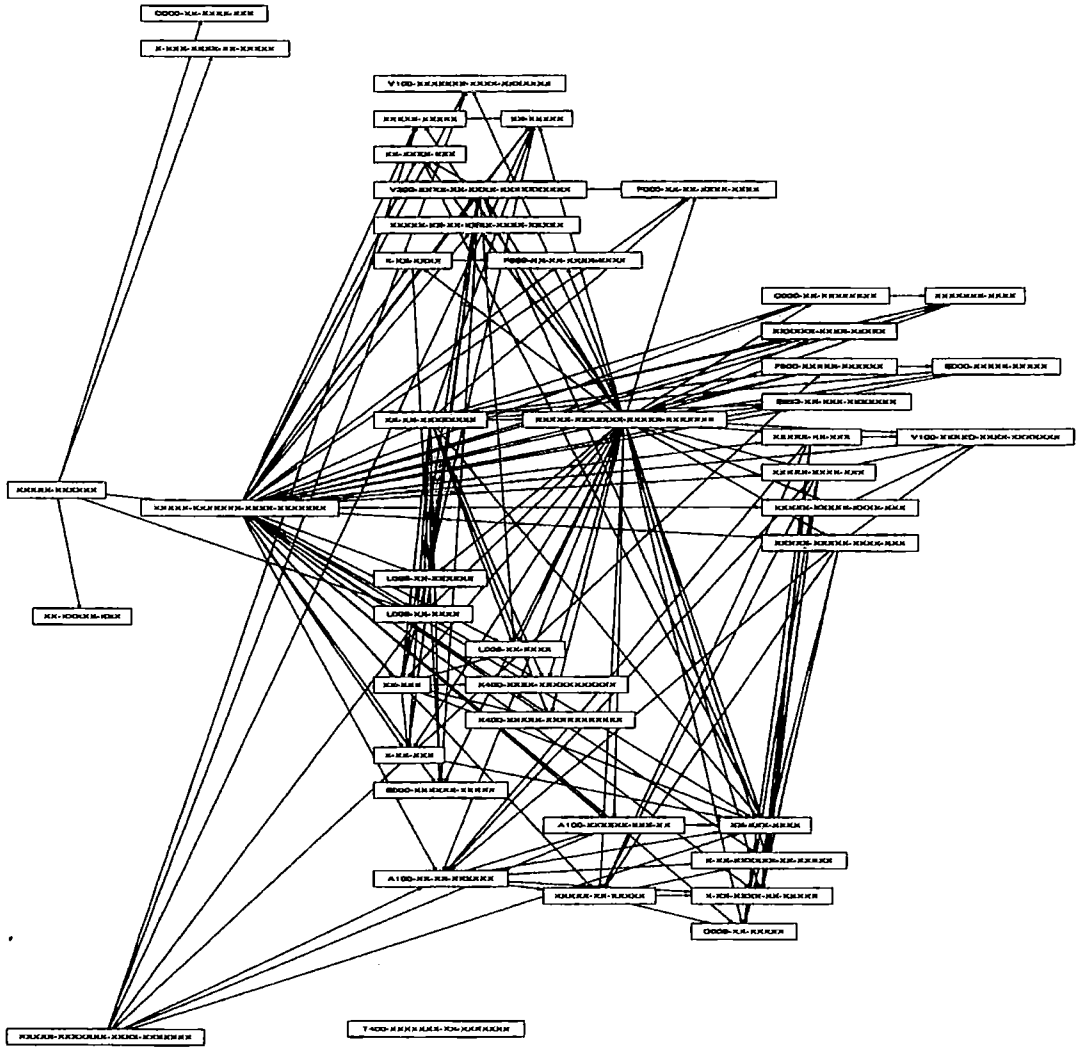


Fig 6.31: V100 a Fully Connected Data Dependency Graph

The result of the data analysis in the T110 SECTION shows a data dependency graph with more promising results. The graph has three main subgraphs (see Figure 6.32) and a number of disconnected data nodes.

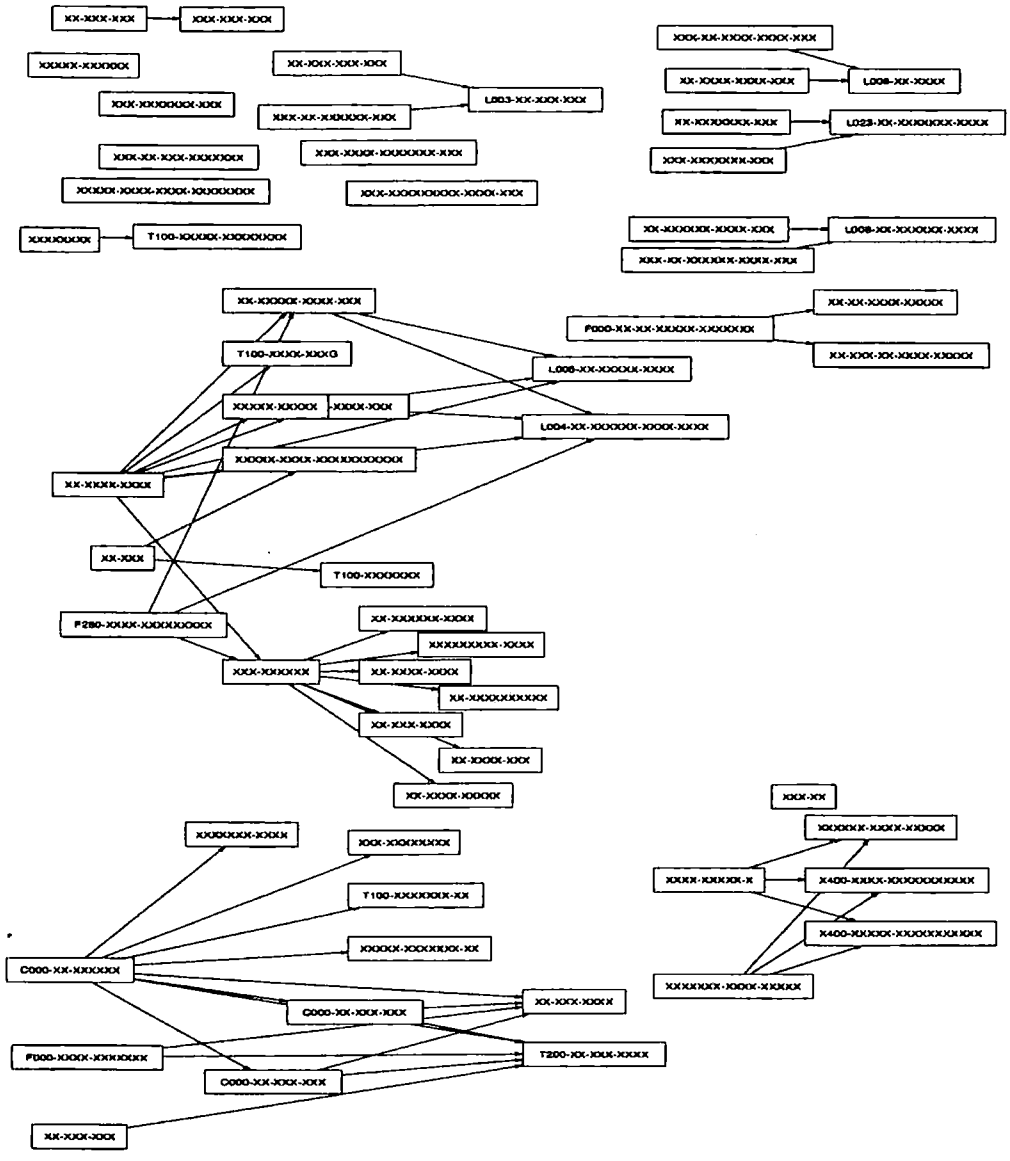


Fig 6.32: T110 a data dependency graph with three main subgraphs

The T110 SECTION is PERFORMed by two other SECTIONs (F090 and T100). In order to slice the SECTION represented in Figure 6.32 it is important to establish if the SECTIONs that PERFORM the T110 SECTION operate on independent data items. If subgraphs of the data are identified, and the data within each subgraph is used by different PERFORMing SECTIONs, then this implies that slicing may be feasible. By comparing the output data of the data items that are used within the T110 SECTION, it is possible to see if slicing the SECTION is a feasible proposition. The results of this analysis have shown that of the three data subgraphs (the three represented at the bottom of the diagram):

- one of the subgraphs data (top left) is referenced only by the F090 SECTION
- one of the subgraphs data (bottom right) is referenced only by the T100 SECTION
- the remaining subgraph (bottom left) is referenced by both F090 and T100.

Therefore, for two of the subgraphs, the results are simple, and the eventual partitioning of the SECTION should be feasible. However, the third subgraph would require reengineering. The SECTIONs called by this SECTION must also be analysed to investigate the SECTIONs output of data. In addition, before a decision can be finally made, the control flow of the SECTION should be analysed.

In the above example, it has been indicated how it is feasible to slice SECTIONs in order to increase the size of reuse candidates. This has been achieved through the analysis of data dependencies. This process has been demonstrated with an example taken from case study C. Within case study B, there were no cycles but there were still a number of directly dominant nodes that may benefit from slicing. One reason why a node is a direct dominance node may be because it provides more than a single function. Therefore, the same data dependency analysis approach can be used for partitioning cycles and direct dominance SECTIONs.

In cases where data dependency subgraphs show data is referenced by more than one of the calling or called SECTIONs, then it may be necessary to consider reengineering the data dependencies. For this approach, the relocation of the data items can be considered as a means of slicing the SECTION. The re-location of data items can also be used to reduce the interactions of data items between reuse candidates. The approach adopted for analysis of potential data items to relocate is that of data usage ranking.

The following example from case study B demonstrates where the relocation of data items is considered to reduce the data interactions. The results of the ranking for the E000 and P000 reuse candidates described within Steps 5 and 9 is as in Table 6.6 from Figure 6.29:

Rank Position	Candidate E000		Candidate P000	
1	23. WS-SXX	106	21. WS-NXXXXX-XXXXX	64
2	14. TS-AXXXXX-XX	34	27. WS-NTXXXXX-XXX-XXXX	37
3	7. I3076-PXXXXXX-XX	8	2. I1071-XXX-XXX	18
4	19. UA-WXXX-XXX-XXXXXXX	6	6. I3076-MXXXXXXX	15
5	18. UA-RXXX-XXX-XXXX	3	3. I1072-XX-XXX	14
6			16. UA-NXXXXX-XXXX-XXX	10

Table 6.6: The Results of the Ranking Procedure and numerical difference of usage

Data items that rank highly within one reuse candidate but achieve a low position within the other reuse candidate are potential candidates for re-location. Those ranked within the top third of all the data items are indicated in the table above. Ranks are identified using total numerical difference of usage between these two candidates (i.e. number of usages for Candidate E000 – number of usages for Candidate P000). In this case, only two candidates are compared so it is only necessary to compare the top ranking data items of each. (Since there are only two candidates and ranking is based on the

numerical difference of usage between the two, those data items that rank highly within one reuse candidate will automatically be ranked a low position for the other reuse candidate.)

The ranking procedure indicates where there is a high degree of manipulation on a specific data item in comparison with the degree of manipulation within its opposing reuse candidate. Therefore, this provides the information as to specific functionality provided by the reuse candidate. Through constant maintenance, functionality may become dispersed. Thus, through this process, an opportunity to reunite the functionality is presented. While each data item ranked highly within one reuse candidate but in a low position within the other (from Table 6.6) may prove worthy of relocation, it is also important to consider the cost implication of this process.

The cost of relocating the data items will be dependent upon the logic in which they are embedded and also the number of data items that are required to be moved. The usages for each data item that would need to be moved to the new reuse candidate are now summarised within Table 6.7.

	E000		P000	
Rank	Name	Move	Name	Move
1	23. WS-SXX	14	21. WS-NXXXXX-XXXXX	4
2	14. TS-AXXXXX-XX	1	27. WS-NTXXXXX-XXX-XXXX	4
3	7. I3076-PXXXXXX-XX	2	2. I1071-XXX-XXX	6
4	19. UA-WXXX-XXX-XXXXXXX	1	6. I3076-MXXXXXXX	1
5	18. UA-RXXX-XXX-XXXX	8	3. I1072-XX-XXX	4
6			16. UA-NXXXXX-XXXX-XXX	2

Table 6.7: The Number of Usages of the Data Item which Require Moving

Comparisons are then made between actual usages (rather than differences) and also how the data items are used within the relinquishing candidate. Relocation is, of course, proposed for smallest usage to the candidate with the largest number of usages. Thus, the process aims towards minimal cost and maximum benefit. The data to begin this process is shown in Table 6.8. The relationships in the 'Use of Relocation Data Item' column (Table 6.8) of the recipient reuse candidate have the data item relationship (R, U) unless otherwise stated. Some of the data items have already been removed from the interface through carrying out Step 9. These are data items 27 and 6 and therefore can be 'crossed out' as they need not be considered.

Rank	Data Item Number	Number of usages E000 / P000	Use of Relocation data item
E000 - 1	23	120 ⇔ 14	(RU)
E000 - 2	14	35 ⇒ 1	(R)
E000 - 3	7	10 ⇐ 2	(U)
E000 - 4	19	7 ⇒ 1	(R)
E000 - 5	18	11 ⇔ 8	(RU)
P000 - 1	21	4 ⇐ 68	(U)
~~P000 - 2~~	~~27~~	~~4 — 41~~	~~(U,U)~~
P000 - 3	2	6 ⇔ 24	(RU)
~~P000 - 4~~	~~6~~	~~1 — 16~~	~~(R, R)~~
P000 - 5	3	4 ⇔ 18	(RU)
P000 - 6	16	2 ⇐ 12	(U, R)

Key

⇒ Input from relocation candidate

⇐ Output from relocation candidate

⇔ Input / Output data from relocation candidate

Table 6.8: Data items considered for re-location and their usage

Those data items with (R,U) relationships are of greatest benefit for relocation as these require both input and output of data. Furthermore, with the exception of a single data item (Data Item Number 16 from Table 6.8) all recipient reuse candidates have (R,U) relationships with the other reuse candidate. Therefore, with the relocation of the data item, both of the sets of data item value passing are no longer required. However, in all the above cases, these would require the most reengineering due to their high instances of use. Therefore, in this case, the most likely candidates for relocation from Table 6.8 are: 7, 14, 16 and 19. Others may be removed as a result of further analysis. However, this will invariably require control flow analysis. The resulting interface between E000 and P000 is shown in Figure 6.33.

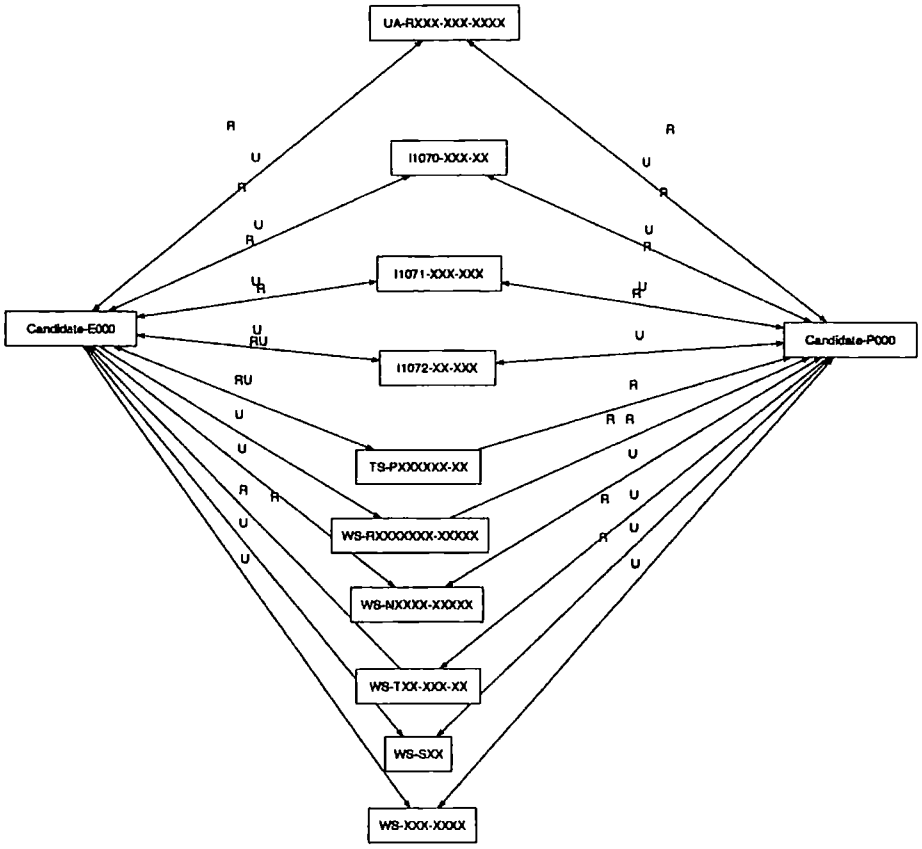


Figure 6.33: Resulting interface of E000 and P000

c) Redevelop if feasible

Thus, the reuse reengineering approach can now begin the reengineering phase where the reuse candidates are actually reengineered for reuse.

6.11. Summary

This chapter has described 3 case studies. The method involves a 10 step process which investigated both the PERFORM structure and the data interactions for COBOL code. The case studies are used to show typical results of the use of this method. In addition, some indication of the suitability of the different sizes of code samples for the application of this method can be gained for the examples presented within this chapter.

Where possible, an indication of the suitability of graphical analysis of the results is given. In some cases, graphical display of results has not been possible, due to the size and complexity of the results. In this case, an alternative textual description of the results has been given. The different sizes of code samples show how each of the approaches deals with some of the issues of scaling.

The method has encompassed both bottom-up and top-down approaches and the results of each of these has been examined and contrasted within the case studies. The results of each of these approaches have been shown to be different, but also compatible.

Chapter 7. Evaluation of the Steps

Chapters 7 and 8 will evaluate the success of the approaches described within this thesis. The evaluation process is carried out in two parts. The initial part, described within this chapter, will concentrate on the usability of the method and detail the costs and benefits of each step of the method. The latter part of the evaluation process will concentrate on the overall benefit of applying the method as a whole. The three main approaches of the method will be examined; specifically the use of dominance trees, cluster analysis and graphical representations. The evaluation of the first two will be addressed by identifying how the results of their application stand up to the process of software evolution and the latter (the use of graphical representations) by evaluating its benefits towards improving program comprehension.

This chapter will now evaluate each of the individual steps of the method. The objectives of each are considered and the necessity of carrying out each step is discussed. The results of applying each step to the case studies are reviewed as well as the provisions made for tool support. Issues such as scaling and tool support provide a critical review of the approach as do the inclusion of issues regarding the step's support for the COBOL language. In addition, any known or identified deficiencies within the steps are reviewed and an indication of how these may be rectified is also given.

Each step of the method is now reviewed.

7.1. Step 1. Generate a PERFORM Graph from the Source Code

The objective of this step is to:

- identify the calling structure of the code on a SECTION basis.

Chapter 4 (Step 1, Task a) describes how the generation of a PERFORM graph based on the calling structure of the code can only be achieved accurately if there exist EXIT points at the end of every SECTION. Furthermore, there should be no use made of GO TO statements between the SECTIONs. Chapter 5 indicated that, in each of the code samples analysed, there were no examples of GO TOs leading to points external to the SECTION and, additionally, there were no problems with fall throughs

at the end of SECTIONs. If, however, these points were not satisfied such problems would not have prevented the use of the method. It would, however, be necessary to perform additional restructuring before the PERFORM graph could be generated. Tools such as Refine [Reasonin] are able to provide such restructuring.

The results of the generation of the PERFORM graph (Step 1, Task b) provide the first indication of the 'structuredness' of the code. The results of the case studies have shown that it is possible to gain an indication of the number of SECTIONs within the code and their interconnectivity. These factors are not based purely on the size of the code, although size is often a reasonable indicator. Graphical representations of the PERFORM graph are a good way of being able to identify the structuredness and complexity of the source code at a SECTION level.

In the examples shown in Chapter 6 it is possible to see a variety of graphs ranging from those with a simple PERFORM structure (Figure 6.1) to those that are far more complex (Figures 6.2 and 6.3). At this stage, a degree of simplification has been achieved for each of the graphs by representing multiple PERFORMs as a number on the arc rather than using multiple arcs. This has been found to make a considerable difference to the readability of the graph. For instance, with the medium sized example case study B (the PERFORM graph is shown in Figure 6.2), this simple procedure reduced the graph from having 992 links to 484.

As the number of nodes and arcs increases on the graph, then the readability decreases. As indicated above, this is not necessarily a factor that is linked to the size of the application (i.e. lines of code), but rather it is a factor of the number of SECTIONs within the code and the number of PERFORMs to different SECTIONs. This is a problem if it is necessary to include these graphs in documentation or they need to be viewed on a computer screen. However, large graphs can be printed in parts for analysis purposes so this is not a limiting factor inhibiting the use of the method.

In Chapter 5 the necessary and actual tool support for this step has been described. Given the structure of the code, excluding fall through and external EXIT statements, the tool support provided was adequate. The one area where improvements could be made is with the graphical representation.

The graphs generated have been laid-out using an automatic layout algorithm. The algorithm tries whenever possible to layout the nodes in a tree structure. This forms a reasonable representation, but at times could be improved upon. The representation as it is, however, clearly places the entry point node(s) at the far left (see Figure 6.2). This provides a useful guide to the location of the main SECTION and is useful information for Step 2.

Despite generation of often complex graphs, the resulting graphical representation is often an initial discussion tool. In general the graph provides a useful overview of the source code and allows the identification of complex areas in the code where there is a high degree of fan in or fan out. These

discussions are useful at later stages of the method where simplification procedures are applied, for instance, with the identification of non-functional aspects of the code (Step 7).

7.2. Step 2. Generate a Dominance Tree from the PERFORM Graph

The objective of this step is to:

- identify the dominance relationship between SECTIONs.

The first task within this step is to remove cycles. The dominance tree can only be generated from acyclic graphs. From the results of the case study, it was found that, with the exception of study B, all of the PERFORM graphs contained at least one cycle. These were removed by grouping all the nodes within the cycle into a single node. Once an acyclic graph was formed, the dominance tree was generated. The results of Task b are shown in Figures 6.8, 6.9 and 6.10. Two types of relationships are shown, those of strong and direct dominance. Chapter 4 indicates how the direct dominance nodes, if they are to be included in candidate reuse units, will require additional modification and restructuring. The strong dominance nodes, due to their simple relationships to nodes only within the candidate reuse unit, can be used with minimal modifying or restructuring of the code.

The benefit of providing this approach is that, for the first time, it is possible to see a relatively simple relationship between individual SECTIONs and those that are PERFORMed by them. These relationships (in the form of subtrees) can be identified at different levels of granularity. For instance, a single subtree may be composed of a number of smaller subtrees that can also be used as candidate reuse units. This is a simple but important stage within the method. A number of candidate reuse units are proposed and discussions can commence as to which of these candidates will be most useful for future developments. The case study has shown a number of varying results after carrying out this step. Case study A showed very few, and small candidate reuse units were identified. Only 5 SECTIONs were included in the largest candidate reuse unit, but this is to be expected when considering the size of the overall code module. Case studies B and C showed more promising results with the dominance tree analysis. Case study B, in particular, identified one reasonably large candidate reuse unit without any direct dominance nodes. From case study C, more candidate reuse units were identified, but these contained far fewer SECTIONs than the results identified for case study B.

The dominance tree analysis is initiated by investigating entry points within the code module. From analysis of the PERFORM graph from Step 1, it can be seen that there is sometimes more than one entry point within a single source code module. This means that a number of sets of dominance tree relationships can be identified from a single code module for each of the entry points within the code. In each of the cases identified as having multiple entry points, it was evident that there was one major, or perhaps 'normal', entry point within the program. For instance, the entry point used most often

within the execution of the program. The other entry points may be concerned with the handling of error conditions. The dominance trees and PERFORM graphs generated from the minor (or error handling) entry points are often represented by very small PERFORM graphs and dominance trees. Often, the other entry points were only composed of up to 5 SECTIONs. An example from case study B is shown within Figure 7.1. In this case the multiple entry points are highlighted as blue nodes; there are 7 entry points within case study B. The red nodes within Figure 7.1 are those nodes which are shared between dominance trees. The existence of more than one entry point within the program demonstrates a weakness in this approach. From viewing a single dominance tree misunderstandings during restructuring could be made with regard to a reuse candidate's composition of SECTIONs. For instance, the independence of PERFORMs within reuse candidates cannot be assumed if its constituent SECTION are also present within other dominance trees.

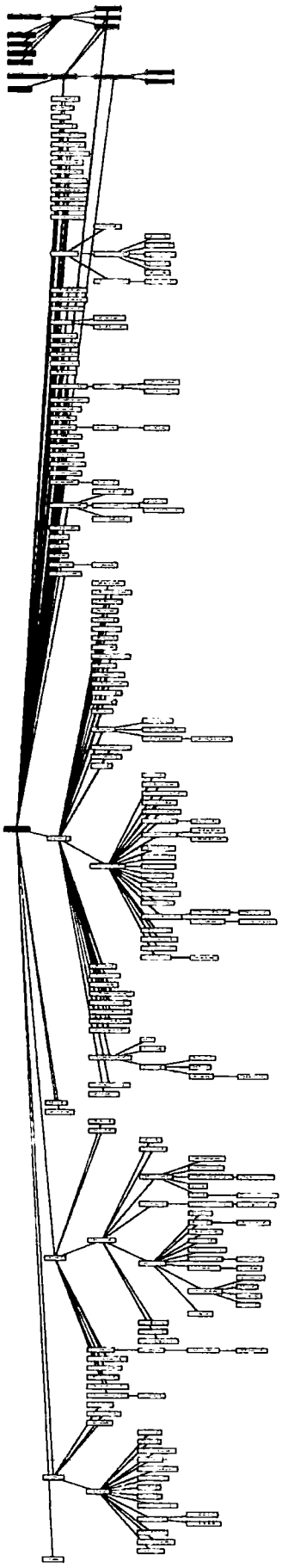


Figure 7.1: Multiple dominance trees sharing common SECTIONs

The presence of multiple entry points within the code has highlighted it is important to consider the other code modules within an application.

The direct dominance relations highlight potential relationships between a possible reuse candidate and the other service candidates or candidate reuse units containing directly dominance nodes. For a potential reuse candidate containing only strongly dominant nodes it is possible to assume that it is not PERFORMed by any SECTION that is represented lower within the dominance tree. For instance, if a reuse candidate has a strong dominance relation between it and the main procedure then this means that the reuse candidate will not be PERFORMed by any other SECTION (other than main). However, what the dominance relation does not express is which of the direct dominance relations within the dominance tree the reuse candidate itself PERFORMs. This information is essential when re-modularising code, however it is equally important that the simplicity of the relations expressed within the dominance trees is not destroyed. A suitable compromise is where, for re-structuring, a new relation is overlaid on to the dominance tree expressing the relationship between a specific reuse candidate and the direct dominance nodes that it PERFORMs. An example of this new relation is shown within Figure 7.2.

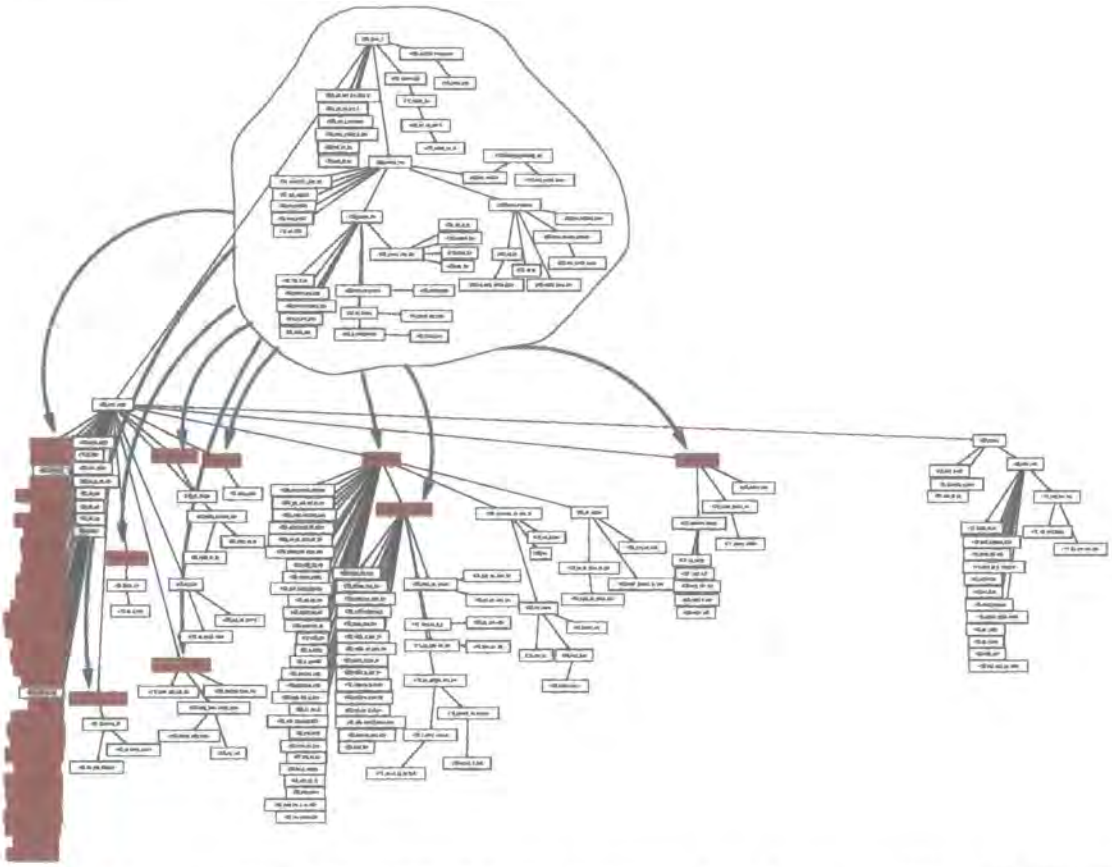


Figure 7.2: Reuse candidate PERFORMs of direct dominance nodes and service candidates from case study B

Within Figure 7.2 a reuse candidate identified from case study B is highlighted in green. The direct dominance nodes and service candidates which are PERFORMed by the reuse candidate are highlighted in red. The proposed new links are expressed as a relationship between the reuse candidate and the direct dominance nodes (the blue links). Many of the SECTIONs within the reuse candidate access the same direct dominance nodes. For instance, 47% of the nodes within the above reuse candidate PERFORM the 'Y400' service candidate. Therefore, expressing the relationships from the reuse candidates simplifies the resulting representation but still provides the necessary information for the re-modularisation process. This new relation provides precise information regarding the PERFORM relationship between the reuse candidate and the remainder of the code. This new representation can be implemented in COBOL during re-modularisation as a CALL. This solution elevates the necessity to increase the complexity of the dominance relation while still providing accurate information for a re-modularisation process.

The size of the source code module was found to have a significant effect on the results of dominance tree analysis. The smallest case study (A) provided a dominance tree with only a single candidate reuse unit and, therefore, highlighted the point that, with such a small initial code sample, there was little benefit in carrying out such analysis. For the larger case studies (B and C), however, there is far greater value in carrying out the method. As indicated in Chapter 6 Section 1, the size of the code module is not always a good indicator of the number of SECTIONs present within the code module. Excluding code modules on their size is, therefore, not a viable approach.

Graphical representation of the dominance trees was of great value when selecting candidate reuse units and showing different levels of granularity. In each of the case studies, directly dominance nodes were shaded. In this way, it was a simple process to carry out the identification of candidate reuse units without needing a detailed understanding of how the relationships were composed. Therefore, this approach means that management can be included within the decision making process without needing a detailed knowledge of the syntax and semantics of the COBOL language. Thus, graphical representations are able to cross the technical / managerial boundaries and allow discussions between different types of staff. Furthermore, with the issues of granularity, it is useful to be able to highlight and annotate the individual subtrees to allow discussions to be made regarding the true functionality of possible candidates. Thus, there is an opportunity to be able to carry out some form of concept assignment during this step.

Tool support provided for this step was adequate for generating the dominance tree. However, there were considerable problems regarding the integration of the independent parts of the prototype, in particular with the translation of the data between the PROLOG path analysis and the graphical display tool. The graphical display tool lays out the dominance tree analysis in a reasonable format, because it is simpler to layout a tree structure then graphs (as identified as problematic with Step 1).

It has been found that this step provides a significant advance towards the identification of candidate reuse units from legacy systems. The approach is simple to use and can be automatically generated from the data provided from Step 1. The graphical representations are also easy to understand and layout is automatic. A considerable improvement of the usability of the approach would be achieved with the full integration of the tools.

7.3. Step 3. Identify Candidate Reuse Units from the Dominance Tree

The objective of this step is to:

- identify all candidate reuse units. Candidate reuse units at different levels of granularity should also be considered.

This step uses the results of Step 2 to identify the subtrees that contain sets of nodes that are strongly dominant (for an example see 6.12). This step is simple to perform when using the trees generated by the prototype tools of Step 2. It is however, possible to analyse the dominance tree automatically, using the same data that is generated by these tools. The results of this step are not final, but rather they must be refined during the later steps. Some of this refinement involves the experts making decisions based upon future business strategies. This is one of the strong points within the method. Users are not blindly lead through the candidate reuse units' selection process. Rather they are expected to drive the process with the aid of the automated support tools.

Potentially, each subtree can be a candidate reuse unit. However, in order for the subtrees to be truly reusable, they must provide a functionality that is useful to the management's future implementation strategy. It would be infeasible to attempt to implement an automated strategy for candidate selection since it relies heavily on human intuition as well as knowledge regarding management's future intentions. In addition, ideally candidate reuse units should provide a single functionality. In order to achieve such a degree of separation of functionality, some form of concept assignment must be established. This is a difficult task that requires a great deal of domain experience. In the case studies, the task was found to be assisted by the availability of graphical representations. Managers were able to indicate areas of functionality based on the SECTION names and the interconnections (PERFORMs) between them.

It is also possible to include in discussions those candidate reuse units which have so far been excluded from consideration, i.e. those with nodes that are directly dominant. It is necessary to evaluate the cost and benefit of using these candidates. This analysis cannot be achieved accurately until detailed data analysis has been completed.

The approach adopted within the case studies was to identify all potential candidate reuse units, some of which were, at later stages, rejected by experts. Candidates that may later be rejected could be those that are considered too costly to reengineer or those that are less likely to be used. The actual decision as to whether they will be developed into reuse candidates should be left until the later stages of the method. In particular until the data analysis is complete.

The studies demonstrated that automatic analysis of the dominance trees was feasible but a far simpler approach was found to be one of visual identification.

A deficiency of the approach should be recognised. Three steps of the method have been completed and at this point it may be evident that there is little value in continuing the analysis. While terminating the method at Step 4 is preferable to being forced to terminate it at the later stages, ideally it would be preferable to terminate the process far earlier. For instance, termination of the process should be at the end of Step 1 or the beginning of Step 2. A simple check included within Step 1 would greatly improve the acceptability of the method. The fact that there seems to be no simple measure to indicate the feasibility of obtaining a range of reuse candidate units means that the inclusion of such a measure is difficult.

7.4. Step 4. Identify Data Dependencies within the Source Code

The objective of this step is to:

- identify which data items are accessed within the source code.

In accordance with Task a, for each SECTION, the following information should be identified:

- which data items are referenced
- whether a data item is Created, Updated, Read or Deleted

In addition, the COBOL constructs which refer to GROUP and ELEMENTARY items are considered along with the consists and redefines relationships (Task b). Note is also taken where pre-set data values means certain data types overlap. Examples identified within the case studies included the use of grouped time periods. The sets of overlapping values are often found to be associated with conditional based statements.

This step is a data collection step that is used to identify the usage of data items within each SECTION. These are later used to compare the interaction or interfaces of candidate reuse units. This is based on the candidate reuse units identified during the previous steps of the method. In order to be able to group

the data items depending on their usage, it is necessary to understand the syntax of the COBOL language.

During the case study, it was found that tool support for this step is essential. The time taken to manually identify and categorise the data items was considerable. The approach adopted within the case study, due to the lack of tool support, was to select specific areas of the source code to demonstrate the concepts. This approach allows specific examples to be highlighted and, in addition, it is feasible to select areas where problems are expected. Although it was found that this was a useful approach, even in relatively small selective areas, the process was time consuming. Furthermore, when full reengineering is to be considered, this selective approach cannot replace the more detailed analysis that will be essential for the provision of accurate cost predictions for reengineering. It was possible, however, to identify each set of consists and REDEFINES relationships. These were described in Chapters 4 and 6.

In some cases it may not be necessary to identify each individual SECTION's data usage but instead to analyse a candidate reuse unit as a whole. The necessity of knowing the data usage of individual SECTIONs is required only when a SECTION is proposed for splitting or relocation (usually within the later stages of the method). However, if tool support were provided, the step's overheads would be less. There would also be benefits in the later stages as there would be no need to repeat the earlier stages where additional data interrelationships were occasionally found later to be important. This makes the overall approach simpler to carry out by making a clearer demarcation between the steps.

Unlike many of the other steps within this method, the effort required to carry out this step is directly proportional to the number of lines of source code within the module. Thus being relatively simply to make an estimation of the effort required to carry out this step.

Graphical representations are used within this step in two ways. These are the consists relationship (see figure 6.17) and the REDEFINES relationship. The REDEFINES relationship is represented with the consists relationship in a combined graph. For the consists relationship colour is used to indicate the level from the DATA DIVISION at which the data items have been used within the code. The approach seeks the existence of relationships between two sets of data items within a single data GROUP hierarchy (in the graphical representation this was represented by two coloured nodes at different levels within the hierarchy). The occurrence of this duel usage means there is a possibility that operations (for instance a write) will modify the value of both data items. The graphical representation of this kind of information means that their identification can be located at a single glance.

The results of the case studies demonstrate that, for the samples considered, it was relatively rare to find cases of the dual usage. However, overlapping values were used considerably. In all cases, these were used within conditional statements and thus form read only data items. This being the case, no special action was needed, but this would not have been the case if write operations had been used.

7.5. Step 5. Identify Data Inter-relationships Between Subtrees

The objective of this step is to:

- identify subtrees which use independent data items.

This step uses the information gained from Step 4 regarding the usage of data items and attempts to identify areas of data independence within the source code (Task a). This provides essential information regarding the possible separation of functionality. The assumption being that, if a part of a program can be shown to operate on a separate subset of data, then it is likely that this portion of the program offers a different and distinct function. The approach is similar to the global based approach described in Chapter 3. However, the approach is aided in this case by the identification of possible candidates prior to this step (from Step 3).

The results of the case study highlighted the concern expressed in Chapter 3 that since COBOL treats all data items as global, the identification of separate areas of functionality was unlikely to be successful using this approach. The approach, using the groupings identified by Step 3, did allow the identification of areas of almost data independence. Within the case studies, the results of this analysis process were very useful in providing information to allow decisions to be made regarding the appropriate level of granularity of candidate reuse units. By investigating the degree of independence of each possible level of candidate reuse unit identification, a decision could be made regarding the level or levels which would require the least amount of reengineering to form candidates reuse units.

The approach used for the identification of areas of data independence is one of cluster analysis (Task b). In particular, Jaccard's coefficient is used. This approach matches parts of dichotomous variables, where each variable has only two values: either the data item is present or absent. Some approaches other than the Jaccard's coefficient would also provide this simple match. However, the distinguishing aspect of the Jaccard approach is that all negative matches are ignored. In this case, negative matches would be the comparison of data items not used within the candidates. This analysis process is useful since it provides an automatable approach for identification of areas of connectivity between the sets of candidates.

Tool support for this initial analysis is simple to provide. An overview of the entire set of candidate reuse units has been represented as a graph (for instance 6.12). This graph provides an outline of the set of data interactions identified between the candidate reuse units. Furthermore, the graph can also be composed of a number of levels, each level representing the data interactions at a different level of granularity. This then provides an abstract view of the analysis process and allows the inclusion of non-technical staff within the discussions of the appropriate granularity level. The case studies

demonstrated that the inclusion of high level management within this process can allow decisions to be made to reflect the future intentions of the company.

Within this step graphical relationships are represented in colour to show the link between SECTIONs and their data usage (for instance 6.18). The case studies identified that, in most cases, the complexity of the relationships meant that an understandable display was unlikely to be achieved. However, considerable use was made of selected representations (where specific portions of the code were represented) to assist with the understanding process. This, at present, is the only realistic use of the graphical representations unless considerable enhancements are made to the display tool to include a number of abstraction mechanisms.

In Step 4, a number of interdependencies between the data items were considered. These included the groupings of data items into GROUP and related ELEMENTARY items and the use of redefinitions of memory locations through the REDEFINES relationships and the overlapping of values. The results of the analysis within the case studies showed that these were not problematic within the code samples. However, this cannot always be guaranteed. These additional relationships can thus be analysed in the same way as the other data items by grouping those related sets and making a separate level of comparison. The best approach to achieve this would seem to be to repeat the initial analysis process by grouping the results of each of the members within the group's set. This can be achieved automatically.

The effort required to carry out this step is dependent upon the number of candidate reuse units that have been identified and the number of data items that are used within them. However, since it is feasible to completely automate this process this is not an important issue. The inclusion of different levels of granularity does provide a more complex representation of the information. However, the decision making process considering an appropriate granularity level is so important that it warrants the inclusion of this additional complexity. Simplifications can be made to the graph by separating out the lower granularity levels into a separate graph. The provision of an overview graph that represents the data interactions at the highest level of granularity with separate graphs representing increasingly lower levels, is a good compromise. The disadvantage of this approach is that the decision maker could potentially be swamped with graphs. However, the results of the case studies have only shown a maximum of four levels of granularity to be identified so it seems that the presentation of too many graphs is unlikely to be a problem.

7.6. Step 6. Identify Potential Reuse Candidates from Users / Designers of the Code

The objective of this step is to:

- apply a top-down approach to the identification of reuse candidates with the assistance of the users and designers of the code.

Steps 1 to 5 have used a bottom up approach that was based on the analysis of the source code. Step 6 is a top down approach. The approach adopted is to seek assistance of expert designers and maintainers of the code with the aim of identifying reuse candidates (Task a). Experts have a free choice of reuse candidates and thus their candidates are not necessarily the same set as were identified during Steps 1 to 5. The experts have available to them, the information that was gained from Steps 1 to 5, thus, presenting then with the opportunity to draw upon this information to aid the decision making process during this step.

The process carried out by the experts is either one of altering existing reuse candidate compositions or identifying new sets. Altering existing reuse candidates can involve either grouping sets of small candidates into larger ones or extracting SECTIONs from a reuse candidate. The approach adopted will depend upon the concept assignment process. The candidates that are identified should be useful for the future implementation strategy of the company. It was indicated in Chapter 5 that the results of the identification process often resulted in small reuse candidates, that is, candidates of perhaps 5 SECTIONs.

The approach adopted in Steps 1 to 5 assumes that, based on the PERFORM structure of the code, a reasonable reuse candidate can be identified. The results of the case studies have shown that this is not always the case. In some cases, identified functionality is split across dispersed areas of the code. Other examples have highlighted occasions where more than one function appears to be implemented in a single SECTION. Within this step, the experts have the opportunity of putting this right (Task b).

No tool support is required to carry out this step. However, the graphical representations generated in the previous steps of this method are of great use to the experts in assisting their decision making process. The case studies have identified that the graphical representations are useful to direct discussions. In particular, the case studies identified that these representations were most useful for being able to group certain set of SECTIONs without having to understand the syntax and semantics of the COBOL language, thus allowing the inclusion of non-technical staff within the decision making process.

7.7. Step 7. Identify Potential Simplification Procedures to Assist Encapsulation

The objective of this step is to:

- investigate heuristics for the simplification of the PERFORM structure and data interactions of the graph to reduce the complexity of its interactions and therefore assist the encapsulation process.

The results of the case studies indicated that certain aspects of the code seem to hold together otherwise separate reuse candidates. From discussions with the experts, these aspects have been identified as often representing, or at least resulting from, the non-functional requirements of the system. The approaches that have been described above are based on grouping of data usage and the PERFORM structure. The nature of non-functional requirements is usually to span across the different sets of functionality of the systems. Therefore, this results in misleading the previous steps of the method into assuming that they are a set part of a specific functionality and thus the previous steps fail to separate unrelated functions. The only exception to this is with Step 6 where experts attempt to split collections of functionality. The information provided by the experts during Step 6, is a useful starting point for the initiation of this step.

The approach adopted within this step would be greatly assisted with a means of drawing upon previous experience. For instance, typical lists of non-functional requirements and how they are often implemented assists their identification (Task a). The accessibility of documentation from previous projects will greatly enhance the usability to the reuse reengineering approach. Furthermore, it can be achieved with very little effort. Lists can be collated over successive runs of the method. Once detailed lists are obtained, it may then be feasible to identify requirements specific to application sets or domains. This information can then be used to focus attention to specific areas and may eventually lead to the provision of at least semi-automated support.

The simplification procedures (see figure6.23) applied (Task e) do not assume that removal is permanent. Rather, the simplification process is a temporary exclusion, to prevent the obscuring of the true functionality. Once reengineering of the reuse candidates is to commence, then these should be reconsidered, as the services they offer are also included within the appropriate reuse candidates. This approach has a great number of advantages especially with relation to improving the simplicity of the identification process. The disadvantage of the approach however, is that it is feasible to forget to consider the costs of reengineering these non-functional aspects when considering the reusability of the reuse candidates.

Scale is a major factor in defining the necessity of this step. With small applications, or those with simple tree like PERFORM structures, there is less need to apply these simplification procedures. In general, experiences from carrying out the case studies seem to indicate that there is a threshold of a number of nodes, and links between these nodes, which define whether the simplification procedures are essential if comprehension processes are not to be compromised.

7.8. Step 8. Isolate Subtree(s) to Form Reuse Candidates using Graph Slicing

The objective of this step is to:

- separate SECTIONs of the code which have been identified as suitable reuse candidates by carrying out the above steps.

This step involves deciding which reuse candidates to select from the code. This decision is made in light of the knowledge gained from the previous steps. The overriding decision point is, however, the expected costs of making the candidate reusable and the amount that the reuse candidate is likely to be reused within the future.

The decision making process is not technical in nature but is dependent upon a company's existing maintenance strategy. The method does, however, provide a number of criteria upon which decisions can be made regarding the future of the proposed reuse candidates. These were found to be of use within the case studies in terms of directing discussions with management.

7.9. Step 9. Identify Data Items in Reuse Units that would Reduce Data Intersections

The objective of this step is to:

- identify where the definition and usage of data items occurs across SECTIONs within different reuse candidates.

During this step, the sharing of data between the reuse candidates proposed in the previous step is examined. The Step aims to reduce the sharing of data where possible, to improve the reuse candidate's interface.

The results of the case studies showed that significant simplifications to the interfaces between the reuse candidates would be made as a result of carrying out this step. (For example, compare Figure 2.27 to 6.29) However, in many cases the interface between the reuse candidates was still more complex than desired. Within the case studies some analysis was carried out to estimate if further simplification could be achieved by relocating data items. A number of simple cases were found to exist. In order to identify more complex cases, it was necessary to carry out detailed control flow

analysis to support this with a reengineering process. Complex tool support is necessary for such a task to be successful and therefore detailed analysis of this step is beyond the scope of this thesis.

In general, this step is very useful as a means of defining the actual data passing requirements of each reuse candidate. The benefit of this step is particularly important within the COBOL language where there are only global data items. Thus, for other languages offering strong typing and the use of local data, the benefits of carrying out this step are less marked.

7.10. Step 10. Identify SECTIONs where Slicing could Assist Separation

The objective of this step is to:

- identify SECTIONs which implement more than one functionality.

This step aims to slice SECTIONs which offer more than a single function in order to increase the size of the reuse candidates. The case studies identified that, in some cases, a number of SECTIONs could be identified which caused the separation of reuse candidates into smaller groups. The results from Step 6 identified that sometimes the reuse candidates identified were smaller that those that the designers thought would be of use. These smaller reuse candidates, however, could often be composed into sets that together constituted the expectations of the designers. This step can be used to rectify this problem.

This step requires a high degree of manual intervention for the identification of suitable SECTIONs and the planning of the slicing process (see Figure 6.31 and 6.32). In addition, however, it is also necessary that this step is fully supported by automated tools for control flow analysis in order to ensure that the separation does not affect the logic within the program.

7.11. Summary

This chapter has looked at the individual steps of the reuse reengineering method. The results of each step's application were examined with reference to the results obtained from the case studies. In cases where deficiencies have been identified within the steps then areas where improvements can be made have been discussed. In particular, this chapter has identified the need for further tool support, specifically in cases where detailed control flow analysis is required.

The following chapter will continue the evaluation process by considering two issues. It firstly considers if the results of applying the method restructures software to a form that is better able to support evolution, thereby reducing the software's legacy properties. Secondly, the chapter considers whether the method's use of graphical representations is a worthwhile aid to the program comprehension process

Chapter 8. Evaluation of Approaches

This chapter will evaluate the three main approaches used within this thesis. These approaches are the use of:

- the dominance relation
- the clustering approaches
- graphical representations

The first two of these approaches will be evaluated by how the results of their applications support the evolution of software applications. As business strategy changes are made within a company, so modifications must often be made to the software to reflect these changes. Changes can effect the PERFORM structure of the code or the data manipulated within the SECTIONs. Through the process of carrying out these modifications, the software is said to be evolving.

Within software maintenance, changes are ideally recorded through version control. Some changes to software will invariably be minor (for instance, a local change that has few, if any, impacts on remainder of the code) although others will be fairly major (for instance, having significant consequences throughout the code). The amount of changes made to a software module is an important consideration because it will dictate the speed of the evolution process.

The aim of this chapter is to exploit the change process by investigating the evolutionary nature of software maintenance in order to evaluate two of the approaches adopted within this thesis. While it is not possible to accurately predict the changes that may be required in future maintenance exercises, what it is possible to do is to best guess these changes by looking at historical changes. In order to be able to assess the benefit of applying, for instance, the dominance relationship approach, the candidate reuse units that would have been generated from an earlier version of the software are identified using the reuse reengineering method. A study is then made to see if these candidate reuse units would have remained intact given the changes made to them within later versions of the software.

The general evaluation procedure used is to apply each approach to the earliest version of the software and then to use each successive version of the software to identify how the process of evolution has occurred throughout the lifetime of the software. The changes required to be made to the candidate reuse unit can then be used to judge the successfulness of the approach. In this way, it is possible to

investigate the suitability of the restructuring process for the candidate reuse units identified. While the changes made in the past will not guarantee the suitability of the restructuring approach, it does give an overall indication of its robustness.

As successive versions were not available for case study A, B or C two new code samples are introduced at this point. These samples are referred to as sample 2 and 17. These samples are also written in COBOL and are currently within commercial use. Each of these new code samples consists of approximately 20,000 lines of code.

The final approach to be evaluated is the use of graphical representations within the method. Evaluation is made by assessing the usefulness of these within the method based on the following assumptions. The assumption is that if, by using the method's first resulting graphical representations (i.e. Step 1's PERFORM graph), accurate decisions can be made regarding the benefit of continuing with the method, then the formation of the graphical representations can be justified. The justification is made on the basis that the graphical representations are providing new and essential information. The evaluation procedure adopted is to investigate if the PERFORM graphs can be categorised accurately into those which are likely to yield low, medium and high numbers of candidate reuse units. If code samples can be visually identified gain little benefit from the application of the method, then the use of graphical representations is advantageous.

Each approach within the method is now reviewed.

8.1. The Use of the Dominance Relation

The suitability of the dominance relation cannot simply be based on trials with differing versions. In order for it to be evaluated fairly, the degree of change for each version must also be considered. The degree of software modification can fall anywhere between two extremes. At the lower end of the spectrum would be a minor change (such as the inclusion of a new SECTION called by a single existing SECTION or the introduction of a new localised data item). At the upper end of the spectrum is a major change to the software. This could be manifested as the complete replacement of a high proportion of SECTIONs or the addition of a number of data items. A fair assessment of the dominance relation, without having to evaluate a large number of systems, are changes that occur midway between these two extremes. For small changes, there are unlikely to be any detrimental effects for the candidate reuse units whereas large changes may equate to almost re-writing the software which will always necessitate some changes to these candidates. However, in the case of medium size changes, it is possible to evaluate the dominance relation to investigate whether it will stand up to normal evolutionary pressures.

8.1.1 The Evaluation Approach

In order to simplify the evaluation of changes on the dominance tree relationships, this section is only concerned with changes to the PERFORM graph. Changes to the use and scope of data items are important and are considered in the following section.

The addition of a small number of additional PERFORM relations between SECTIONs can make a significant difference to the dominance tree, and therefore to the candidate reuse units that are selected. The difference depends upon where, within the tree structure, additional relationships are added. Therefore, it is not simply the size of modification that is significant, it is also the type of change that is taking place. This section concentrates on typical changes and investigates how these affect the evolution process. Some examples of typical changes that may occur are given in Figure 8.1.

For the purpose of this example, the PERFORM graph with the resulting dominance tree below (Figure 8.1) is assumed to be the starting point before modifications are made.

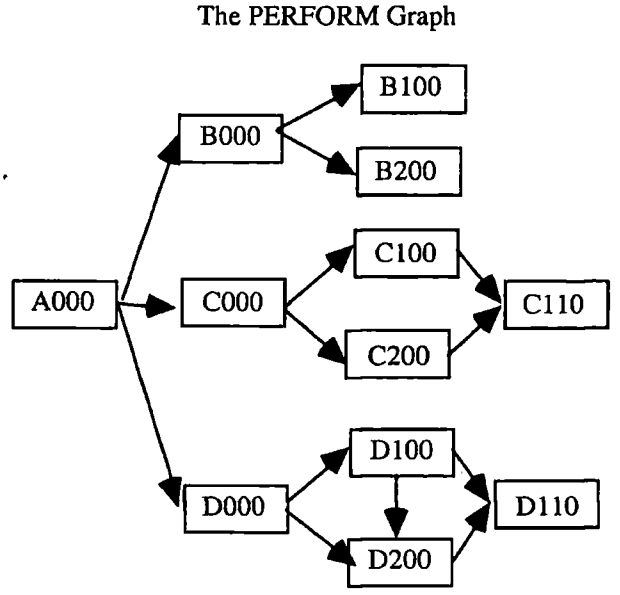


Figure 8.1a: A PERFORM Graph

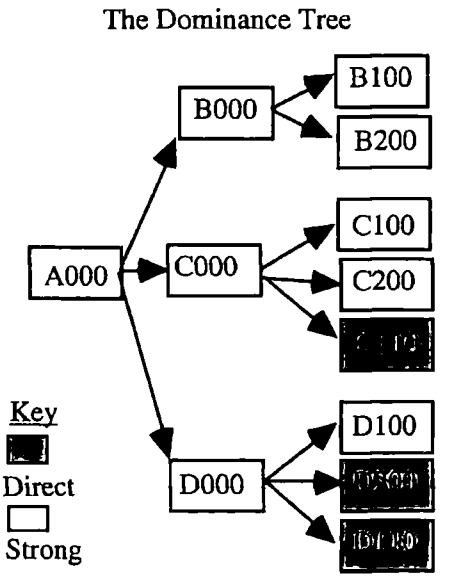


Figure 8.1b: The dominance tree

In the examples below, the effect of the addition of specific relationships to the PERFORM graph is shown. The first of these examples will be the addition of a new PERFORM between procedure B000 and C110. The resulting graph structure and dominance tree are shown in Figure 8.2.

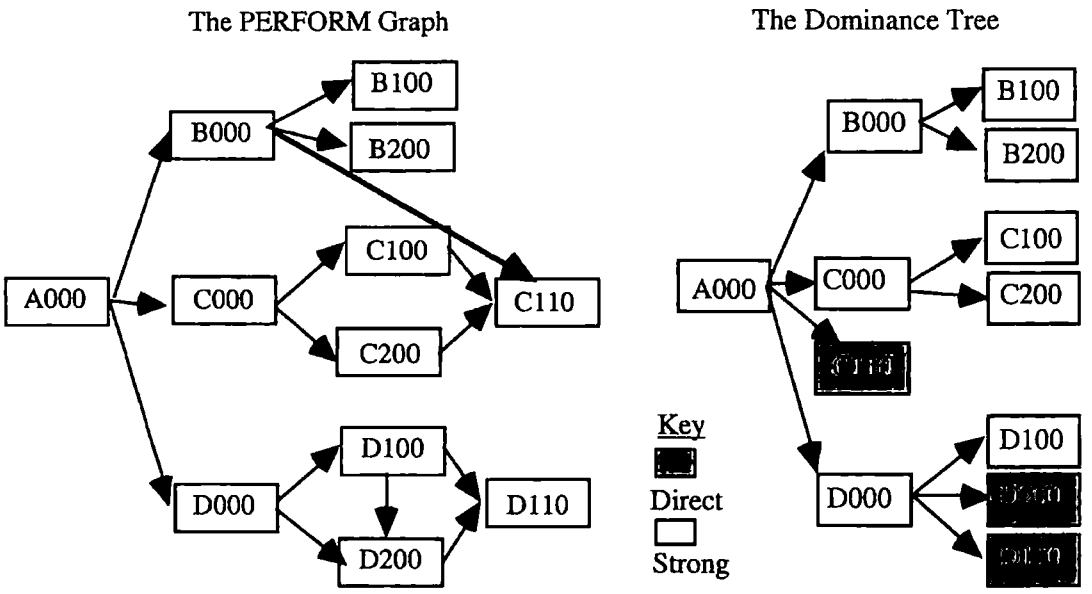


Figure 8.2: Modifications to PERFORM graph and the resulting Dominance tree

The result of this modification means that the candidate reuse unit has one fewer SECTION than in the original example. In addition, a new 'service' candidate has been generated. A service candidate is a single SECTION that can be accessed by one or more of the candidate reuse unit. In this case the service candidate will be PERFORMed by procedure B000 and C100. The consequence of such a change is not too severe as it does not require a great deal of reworking of the candidate reuse unit's construction. Candidates B000 and C000 will require access to the service component (C110) but execution will not switch between the individual candidates. An example where changes in flow of control can occur is given in the example below. The addition of new service candidates is a natural part of the evolution process. This often results from the formation of new areas of functionality. Such a change cannot easily be predicted and should not be prevented since it represents a form of evolutionary progress. Therefore, it cannot be seen as a failing of the dominance tree approach.

The second example shows the addition of a new relation between procedure B000 and C000. The resulting PERFORM graph and dominance tree are shown in Figure 8.3.

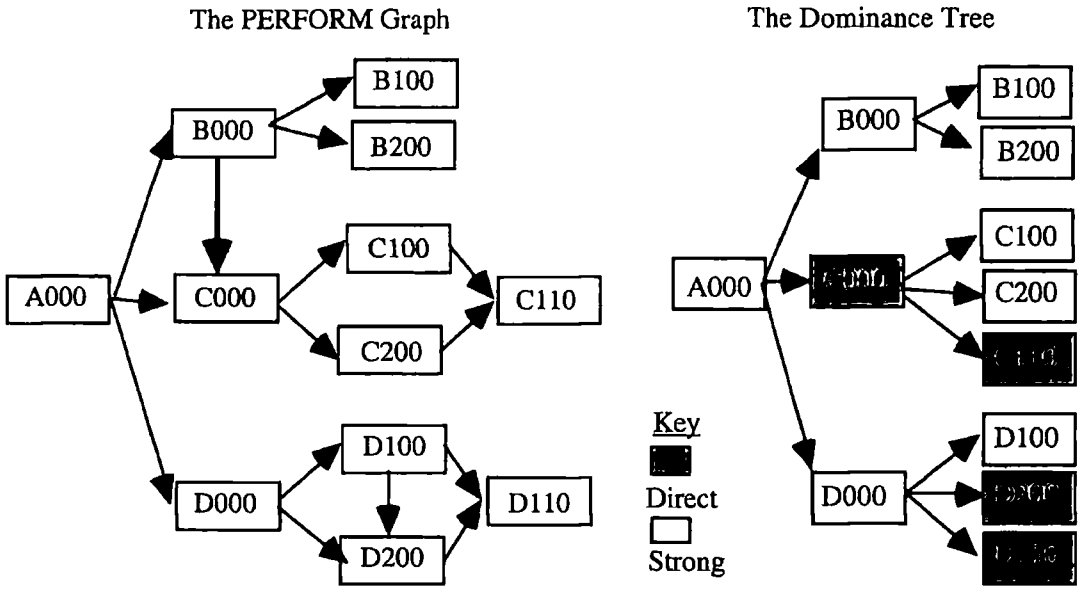


Figure 8.3: Modifications to PERFORM graph and the resulting dominance tree

The result of this modification is that the candidate reuse unit C000 becomes directly dominant upon B000. The consequence of this change is that, upon the execution of procedure B000, execution could pass to the remaining nodes within the original B000 tree or now, in addition, could switch to the C000 candidate reuse unit. The result of this change is that no longer can these two candidates be considered as independent, and in this case it is necessary to reconstruct the two candidate reuse units into a new single candidate. Thus, the result of making this change has severe consequences by creating the necessity for much reworking. Furthermore, the result of this change, unlike the first, could potentially indicate a failing of the dominance tree approach for the identification of candidate reuse units. This is because the approach has appeared to fail to identify a link between the two separate candidate reuse units. Thus, a failure to properly isolate a specific and entire concept has been identified. The consequence being that the link between the two has later been identified by the addition of the new PERFORM.

If, during the process of evolution, it is necessary to perform many of the serious types of changes, then a reasonable assumption would be that the dominance relation is not supportive of software evolution. In order to evaluate the suitability of the dominance relation, a number of case studies have been carried out and are described in the following section.

8.1.2 The Results

In order to evaluate the suitability of the dominance relation, a number of case studies have been carried out. Further case study examples are introduced at this point within the thesis. In this case, code samples are selected which have successive versions of the code available. The selected applications

are, however, taken from the same sample application set as the case studies presented earlier. These are also written in COBOL and produced using the same coding standards.

For each of the source code modules the following tasks were carried out:

1. Produce a PERFORM graph based on the PERFORM and SECTION relationships
2. Produce a dominance tree based on the PERFORM graph
3. Identify code samples with medium sets of changes (i.e. where changes have been made to the PERFORM structure and where a consistent naming convention has been used between versions)
4. Resolve difficulties (i.e. where SECTION names have been changed due to the relocation within the PERFORM graph of the SECTION)
5. Identify any differences between the relations between SECTIONs

The results of carrying out these tasks are now described.

For each of the code samples, the same pattern of results has been identifiable. The results of the analysis process of two samples of code are indicated in detail. These are referred to as sample 2 and sample 17.

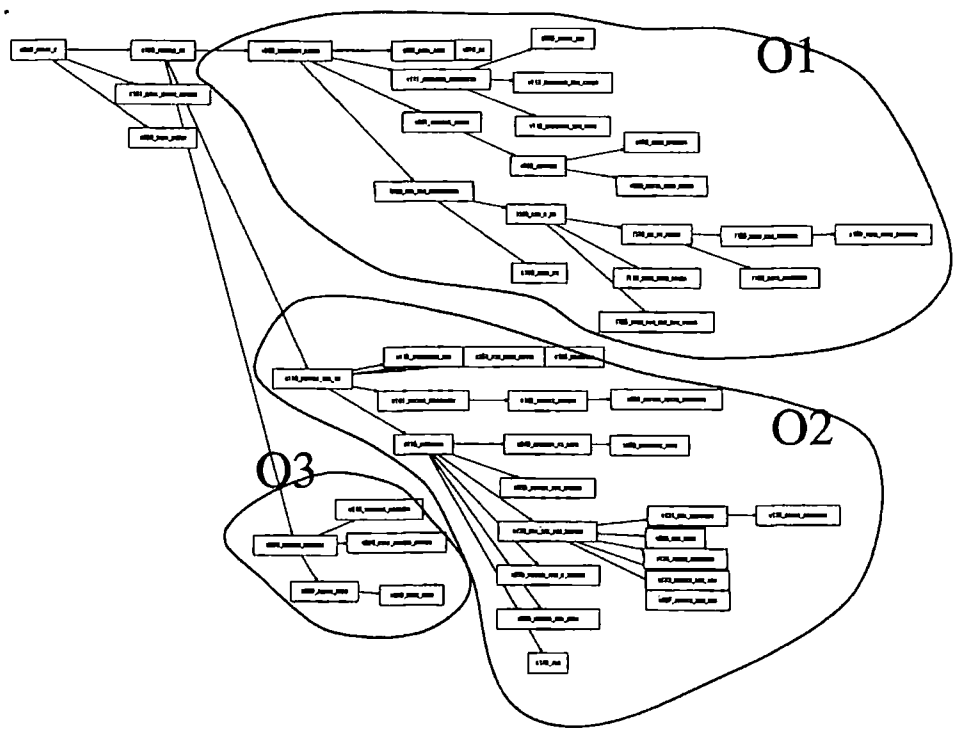


Figure 8.4: A single reuse candidate showing other possible levels of granularity for code sample 2

Figure 8.4 shows a single candidate reuse unit which was identified through the use of the dominance relation in an early version of the software of code sample 2. This candidate reuse unit consists of 50

SECTIONs. Candidate reuse units at lower levels of granularity are also available. The candidate reuse unit within Figure 8.4 could have also been separated into 3 candidates circled and marked O1 to O3 on the diagram. These three candidates would have then consisted of sets of 20, 21 and 5 SECTIONs. The larger candidate (all of Figure 8.4) was selected because some of the candidates (i.e. O3 was only five SECTIONs) were considered too small to be suitable for reuse and concept assignment.

Within Figure 8.5 the reuse candidate from Figure 8.4 is shown to the left. To the right is represented the resulting code after the maintenance interventions. Some parts of the calling structure have remained in tack others have changed significantly. In order that the changes can be mapped outlines are marked around portions of the original candidate reuse unit. Each of these outlines represents a newly formed candidate reuse unit in the later version.

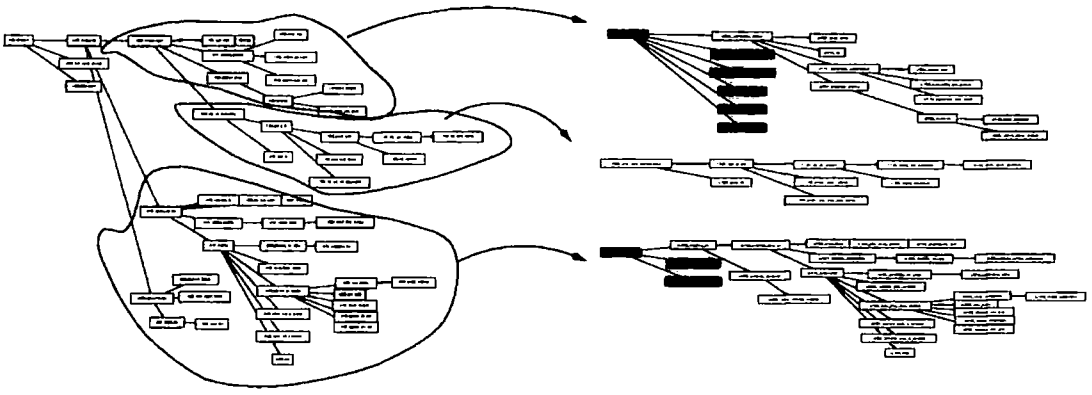


Figure 8.5: The result of the evolution process on the reuse candidate from Figure 8.4

Re-running the dominance relation on a later version of the software identified 3 candidate reuse units from the single candidate identified earlier. This process is represented in Figure 8.5. The new candidates' formation, in relation to the three candidates available at the lower level of granularity, are equivalent to candidate O1 being split and the remaining two (O2 and O3) being joined. Furthermore, a number of new nodes have been added to the candidates. These are represented as shaded nodes on the right hand graph of Figure 8.5.

The candidate reuse unit labelled 03 in Figure 8.4, is grouped within the bottom candidate reuse unit (within Figure 8.5) of the later version of the software. One node in this group has been removed, i.e. the SECTION has been deleted, but two still remain within this part of the dominance tree. The remaining two nodes exist within the code but are not present within this dominance tree. Instead they form another entry point for the program i.e. the PERFORMs to these SECTIONs have been removed between versions. In this case it is likely that the remaining PERFORM structure present within the code (i.e. between the two nodes) now represents dead code within the module.

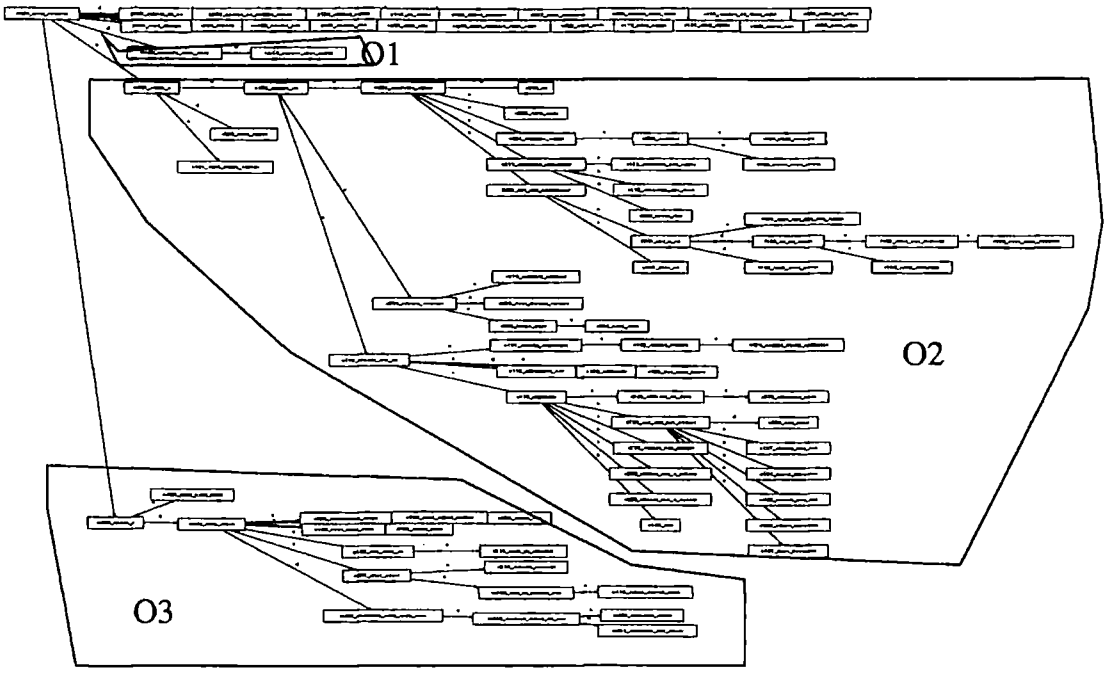


Figure 8.6: An entire dominance tree for code sample 17

For the second example an entire dominance tree is investigated using code sample 17. The first version analysed led to the identification of three candidate reuse units. These consisted of 2, 51 and 18 SECTIONs. These are labelled O1 to O3 in Figure 8.6. Candidate O2 offers differing levels of granularity and could be decomposed into 3 smaller candidates. The other candidates cannot feasibly be decomposed into smaller candidates. Again, by investigating a later version of the code, the candidate reuse units present in the later version have been identified. 7 candidate reuse units were identified in total. These can be seen in Figure 8.7. Comparing the versions of the software, it was identified that O1 remained unchanged. However, O2 was later split into three candidates (labelled O2.1, O2.2 and O2.3), candidate O3 was also divided into three. The addition of 12 new SECTIONs has been identified through the comparison of the versions. In addition, a single SECTION has been removed and, as with the example above, some evidence of dead code has been found.

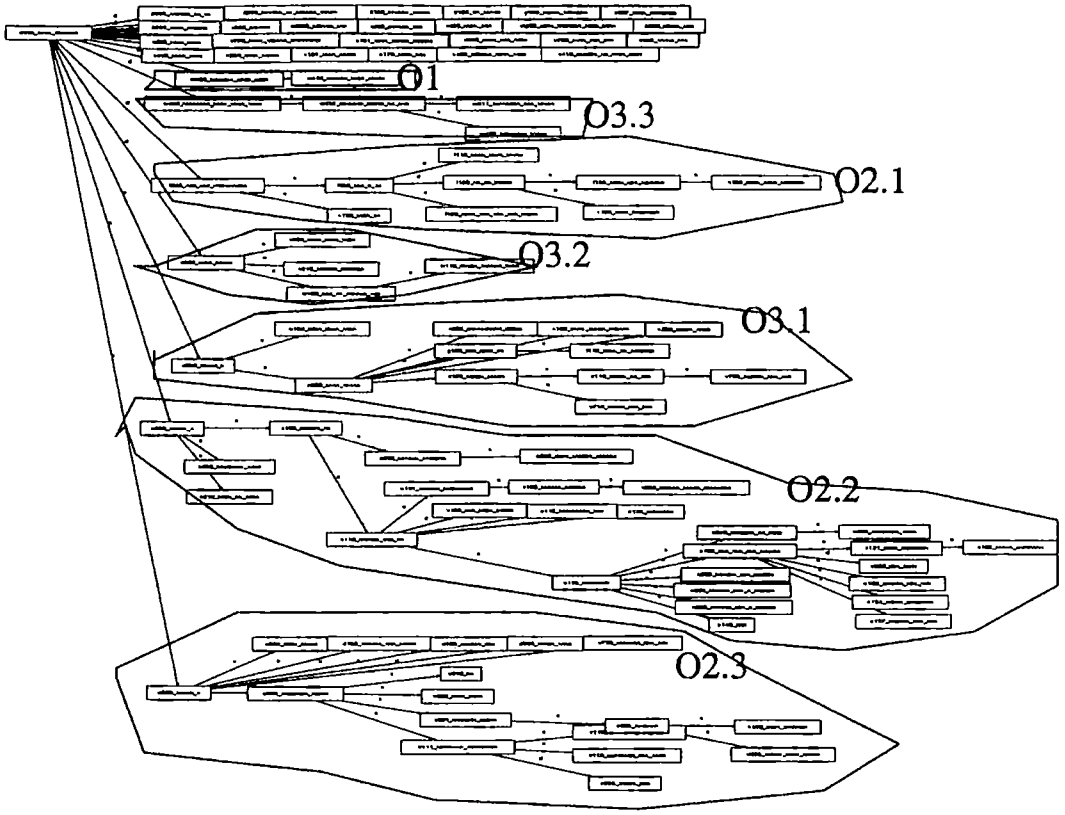


Figure 8.7: The new reuse candidates from Figure 8.6's dominance tree of code sample 17

A failing of the dominance approach was potentially when SECTIONs within a particular candidate became relocated to other candidates during the process of evolution. In order for this situation to occur, the initial version of the software must contain more than a single candidate reuse unit. No examples of the movement of SECTIONs between candidate reuse units were identified in the subset of systems analysed. However, there were occurrences identified of SECTIONs becoming service candidates as was demonstrated by the example within Figure 8.2. Two examples of this were identified in the code sample within Figure 8.7. The service candidates within Figure 8.7 are those unbounded into groups.

New candidates seem to be formed from the splitting of original candidate reuse units, particularly when additional SECTIONs have been added to the code samples. When considering the lower levels of granularity identified for candidate reuse unit O2 (Figure 8.6), findings showed that the same result occurred with this code sample as for the initial example (Figures 8.4 and 8.5). In this case, one of the lower levels of granularity has been split into 2 candidate reuse units, whereas the remaining candidates have joined to form a single candidate. However, since the higher level of granularity was selected by the method in real use, there was no joining of candidates. In this case, the same problems as identified with the example within the previous section are not evident.

Until this point, only the changes to candidate reuse unit's composition have been considered rather than changes to the individual dominance relations. Although the higher level view given provides a

good overall indication of the evaluation process, it is still important to consider individual changes when evaluating the suitability of the dominance relation. For this reason, an overview of the changes to the strong and direct dominance relations is now given.

Changes to the direct dominance relation can effect the constituent nodes of a candidate reuse unit. This occurs in four main ways. The evolution process can lead to a strong dominance relation becoming a direct dominant one. This results from the splitting or grouping of candidate reuse unit or the creation of new service candidates. In addition, the evolution process can also mean that the reverse of this change occurs, for instance, where a direct dominance relation is converted into a strong dominance relation. Further changes that may be expected throughout the evolution process involve the movement of an existing direct dominance relation to a higher position within the tree (i.e. the creation of a service candidate) or the addition of a new direct dominance relation through the inclusion of a new node.

The results of the evolution process on the dominance relation for the evolution case study code is shown below.

	No. of occurrences	
Type of evolutionary change on dominance relation	Example 1	Example 2
Change from strong to direct dominance	2	3
Change from direct to strong dominance	0	0
Still direct dominance but moved location within tree	3	3
Direct dominance by addition of new node	1	2

Table 8.1: Types of evolutionary change

In each of these cases the change in direct dominance relations results from a specific location of the dominance tree structure. Specifically, this is usually where splitting of the tree structure represents the creation of a new candidate reuse unit. Examples of likely candidates are highlighted with arrows within Figure 8.8.

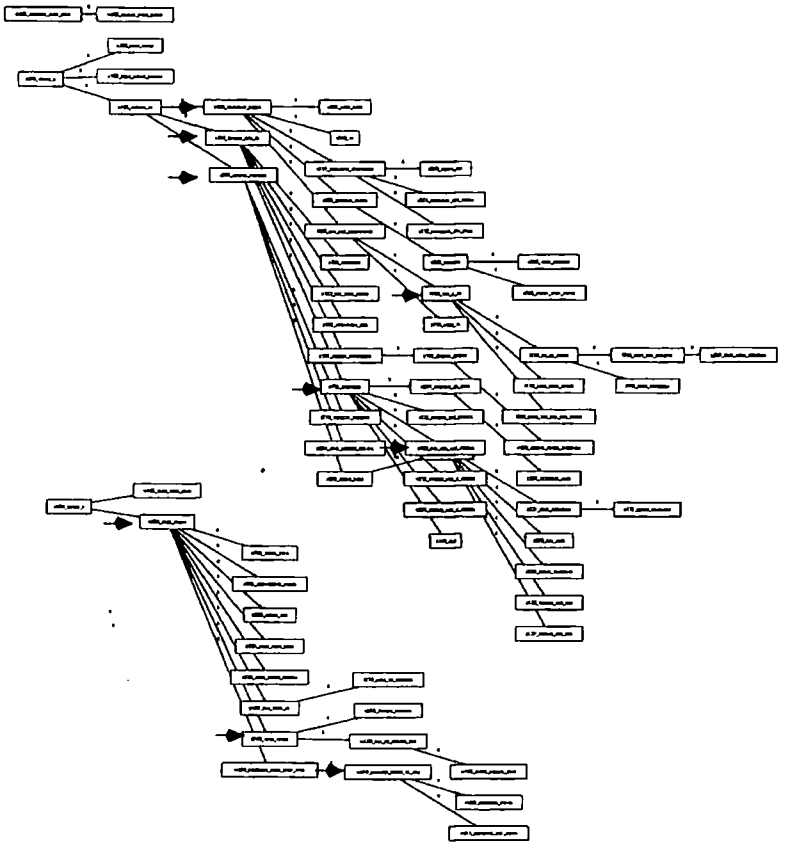


Figure 8.8: Potential portions of dominance tree where splitting is possible

8.2. The Use of Clustering Approaches

Data clustering is an approach to re-modularising software systems. It is based upon the assumption that the data drives the functionality of the program. Thus, if subsets of data can be derived, these should represent boundaries of specific functionality. A number of data structuring approaches were described in Chapter 3.

The clustering approach used within the method is based on pre-selected potential candidate reuse units identified from the application of the dominance trees. This approach is termed an assisted approach as candidate reuse units are not formed purely on the basis of the clustering approach. However, in order to evaluate this approach both an assisted and non-assisted approach (where clusters are based purely upon the results of data clustering) are evaluated to see how the results obtained are affected by the process of evolution.

8.2.1 The Evaluation Approach

To gain an understanding of the evolutionary changes that have occurred to each of the applications within this study, the following tasks have been carried out.

1. Identify the PERFORM structure of the specific version
2. Identify the data items listed within the data division
3. For each SECTION, identify the data items used and the type of usage (i.e. whether it is a read or write operation).

Using the above tasks the successive versions of the application are then compared. The following information is recorded.

1. Changes to the PERFORM structure:
   - as carried out in previous section
2. Changes within the DATA DIVISION, including:
   - new additions / deletions of data items
   - movement of data to or from WORKING STORAGE / USER AREA
   - changes in the data definition (i.e. changes in size)
   - changes in its COBOL level (i.e. 05 -> 01)
   - minor changes in data item's naming (i.e. EXTN-NO -> EXTENSION-NUMBER)
3. Changes within the procedure division, including
   - addition or deletion of data items
   - changes in the usage (i.e. read to read and write)

By collating this information and then comparing the results across different versions, it is possible to identify the process of change that has occurred. In this way, it is possible to begin to understand exactly how software applications' data evolves. Furthermore, by mapping the actual process of change to the results of a data clustering process, an understanding of the suitability of data clustering as a reengineering approach can be gained. This section, using the techniques described above, will seek to investigate the process of evolution and the suitability of data clustering.

One of the less dramatic approaches to the reengineering of source code is to, where possible, keep the COBOL SECTIONs relatively unchanged. In this case similarity measures between data items are based on groupings of procedures using specific sets of data items. Thus, the technique of restructuring would be to re-group sets of SECTIONs that access the same data. For this reason it is important to understand the process of change occurring to data items within each of the SECTIONs. This, however, necessitates the need to firstly study the PERFORM structure of the code. The results of this process have been described in the previous section.

If the clustering approach is to yield successful results, then there are a number of factors that must be considered. These are:

- The number of data items which are localised within a candidate reuse unit. An increase in data items that are localised within a proposed candidate reuse unit will inevitably increase the likelihood of a simpler interface between the other candidates. The success of the clustering approach can therefore be identified by this measure.
- The simplicity of the interfaces between candidate reuse units. Clear and simple interfaces between candidates are those with fewer data items whose usage is shared between them. Ideally the relationships should be as simple as possible, for instance, by having a small number of specific relationships between set numbers of specific candidates.
- The candidates reuse units identified from the approach should be of a size suitable for reuse. Although it is difficult to give specific guidelines regarding optimal sizes of candidates, in general, very small candidates are not worth reusing whereas candidates which are too large are often difficult to reuse (mainly due to the costs of program comprehension for the component).
- The amount of changes in a SECTIONs' composition for specific candidate reuse units during evolution. As with the dominance trees, candidates that change the least are preferable since little re-work be required for the application to evolve.

These factors are now considered in relation to the results of the evolution case study.

8.2.2 Results of the Analysis of Data Changes

Evolution within Code Module 2

In code sample 2, a total of 3067 data items were identified. Through the study of different versions of the software, 922 of these data items were found to change in terms of their presence or absence within a specific SECTION. In this case, only gross changes to the data usage have been accounted for. Subtle changes regarding their use within the program logic are not included. Even so, still over 30% of the data items were found to change within 4 releases of the software. Of these changes, 38% were due to the addition of new SECTIONs, whereas only 7% of the changes were due to the removal of SECTIONs. Furthermore, the effects of the changes were distributed throughout the code. Of the 117 SECTIONs that were present in the last version studied, 66 of these SECTIONs had gross changes in their data usage.

Evolution within Code Module 17

For code sample 17, it was found that 4142 data items were present within the code sample, of which 42% were found to change over time. In this case two distinct changes were made. One involved the addition of new functionality, and the second, the process of preventative maintenance. In particular, the preventative maintenance change involved ensuring that coding standards were still adhered to which may have been violated due to the effects of the other change. 19% of the changes made corresponded to the process of preventative maintenance. In addition, 9% of the changes were accounted for by the removal of SECTIONs, whereas only 1% were due to the addition of SECTIONs. Due to the changes, a modification to one data item impacted changes to 27 SECTIONs.

Assisted and Non-assisted Approaches to Clustering

The modifications made to SECTIONs during the propagation of data changes have an effect on the clustering process. When carrying out cluster analysis, it is possible to opt for either an assisted or non-assisted approach. An assisted approach is where other decision making factors are used as guides for the initial clustering. In the examples below, an assisted approach is selected. This approach bases clusters upon the objects identified from the dominance trees. The code modules from the case study have shown that, in early versions, fairly simple relationships result from the clustering approach. Simple relationships are characterised by the small number, and distinguishable sets, of data shared between the candidate reuse units. A good example from code module 2 is shown to the left of Figure 8.9. In the figure, the candidate reuse units are represented by the small shaded boxes. The larger boxes linked between the candidate reuse units are the data items.

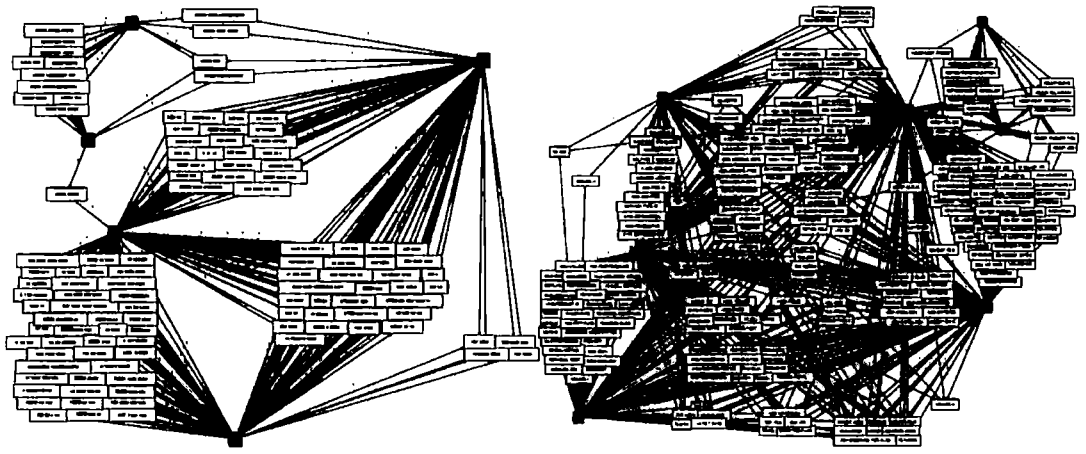


Figure 8.9: Overlap of data usage between candidate reuse units

The results of the dominance tree analysis have indicated that, as the process of evolution progresses, the candidate reuse units identified gradually split. The result of this process is that the complexity of the relationships between the candidate reuse units increase in terms of the data shared between the

new candidate reuse units. This process can be seen in the representation of the same software module (to the right of Figure 8.9) but represented at a later stage of the evolution process. This increase in complexity could be caused by a number of factors, for instance:

Factor 1 The new candidate reuse units may have only been formed due to changes in the PERFORM structure and since no changes were made to the overall functionality (hence the data used) splitting the candidate reuse units will not have decreased the amount of data interactions between the original candidates.

Factor 2 The wrong approach to clustering may have been selected. The poor results obtained from the inter-connectivity of candidate reuse units may mean that an incorrect selection of clusters has been made.

Factor 3 The process of performing evolution may have increased the complexity of the interactions and as a consequence reduced the distinguishability of the clusters.

The remainder of this section will discuss the likelihood of each of these factors affecting clustering results.

If *factor 1* was to be valid, by reforming the original candidate reuse unit structure (or as near as possibly), it should be feasible to see the simple data clustering return. The re-formation process is achieved by grouping candidate reuse unit based as near as practicable to their original constituent groupings from the initial dominance tree. Figure 8.10 shows an attempt to achieve this. In this figure it is possible to see that the number of data items which are interfacing the candidates has been reduced. However, the original distinguishability of the sets of relationships has not returned. Therefore, as a consequence, it is possible to assume that this factor, *factor 1*, is not the only contributor. It is therefore likely that the process of evolution has a considerable effect on the data.

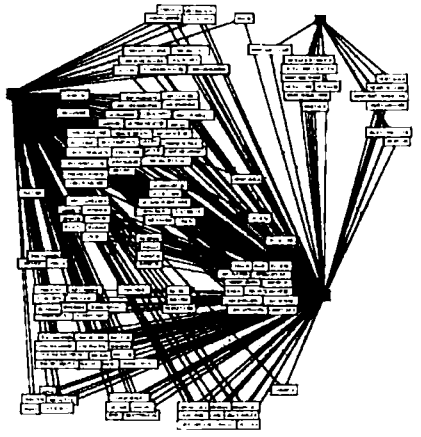


Figure 8.10: An attempt to reform the object clusters

In order to establish the effect of *factor 2*, a different approach to clustering is selected. In this case a non-assisted data clustering strategy is used. This approach to data clustering has the opportunity to reduce the data complexity between the clusters by moving SECTIONs to other groups where the

overall result is a decrease in data interactions between the clusters. However, the results, shown in Figure 8.11, seem to indicate that the results of this approach are similar to the assisted approach. In Figure 8.11 the SECTIONs are the shaded boxes. Data is unshaded. The complexity of the interactions between candidate reuse units (sets of grouped SECTIONs) is still quite high. Additionally, far more and smaller candidate reuse units are defined using this approach. Thus, it seems that it is not the approach to clustering that is at fault so the complexity of the relationships between the candidates is at least partly due to the evolution process, *factor 3*.

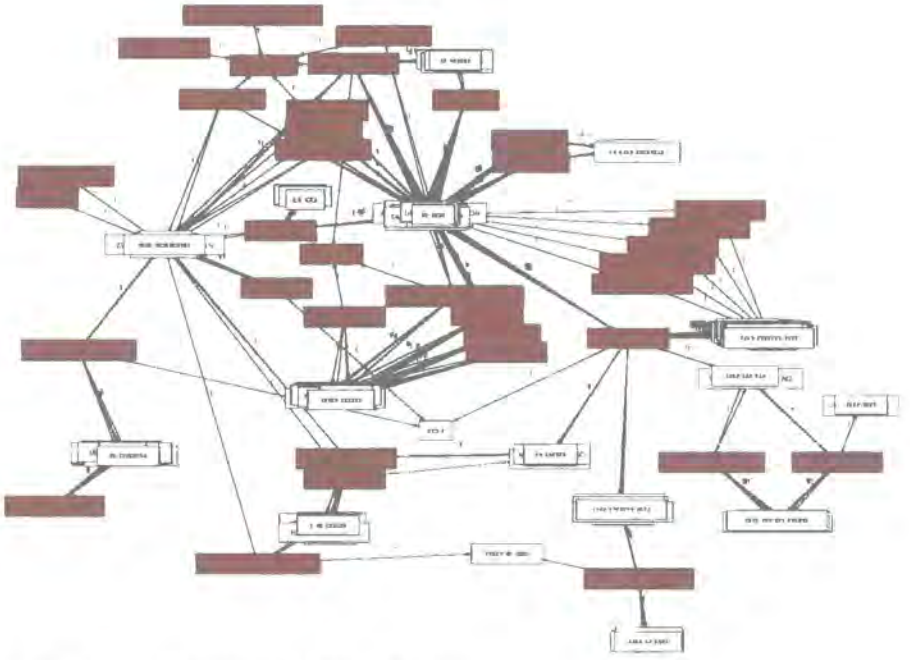


Figure 8.11: Non-assisted clustering approach

In order to evaluate the effect of evolution on data, this thesis now considers the effects of change reachability. Change reachability is the term that expresses the direct consequence of changes to a single data item in terms of the SECTIONs involved within a change. Thus, this term is more restrictive in its scope than impact analysis. An example of a large change reach may be the addition of a specific data item to a number of different SECTIONs.

The reach of a change to a specific data item is an important factor to consider when evaluating the suitability of data clustering as an approach to reengineering. From code module 17, it was found that a single data item change may span 27 SECTIONs. This would indicate that a clustering approach needs to retain each of these 27 SECTIONs within a single cluster. If all the relationships between the other data items in each of the 27 SECTIONs are also considered, the result could be that very large clusters are obtained. However, this in itself should not be seen as a failing of the approach. Rather if the similarity (in relation to the data which they use) of the SECTIONs are great, there may simply be no necessity for re-modularization.

However, it is important to make a distinction between the closeness of a set of SECTIONs due to their provision of similar functionality, and the necessity to make changes to many SECTIONs because of the poor structure of the code, possibly due to prolonged maintenance. In order to test this hypothesis the change to specific data items across successive versions of the software are considered. In particular, how the reach of the data item is affected over time is interesting and a good indicator as to whether the complexity of the data interactions are changing over time. A data item that can be seen to be gradually used by more and more SECTIONs will mean that the clusters obtained will become correspondingly larger. If many data items within an application all show this same trend then the suitability of a data clustering approach to reengineering is reduced.

The chart in Figure 8.12 shows the overall changes for code module 2. The '0' point in the graph represents the data items within the specific portion of the code which have remained unchanged. Those to the left of this point represent data item usages within SECTIONs which have been deleted through the evolution process, and hence, represent places where increased localisation of the data is achieved. To the right of the '0', data items which are new to specific SECTIONs are shown. The addition of existing data items within other previously unaffected SECTIONs describes the data de-localisation process. In terms of maintenance, as well as the suitability of the data clustering approach, the localisation process can broadly be seen as good, whereas the de-localisation process is detrimental to future evolution. From the graph it can be concluded that in most cases changes of -3 or + 6 are common. This can be interpreted as 15 data items have been removed from up to three SECTIONs (-1 =5, -2 = 4, -3 = 6) and that 63 data items have each been added to up to 6 other SECTIONs. One of the data items has been added to no fewer that 39 SECTIONs. These facts seem to indicate that the complexity of the data items is steadily increasing through the evolution process. Thus, the trend is generally detrimental to the maintenance process as the software seems to be acquiring further legacy qualities.

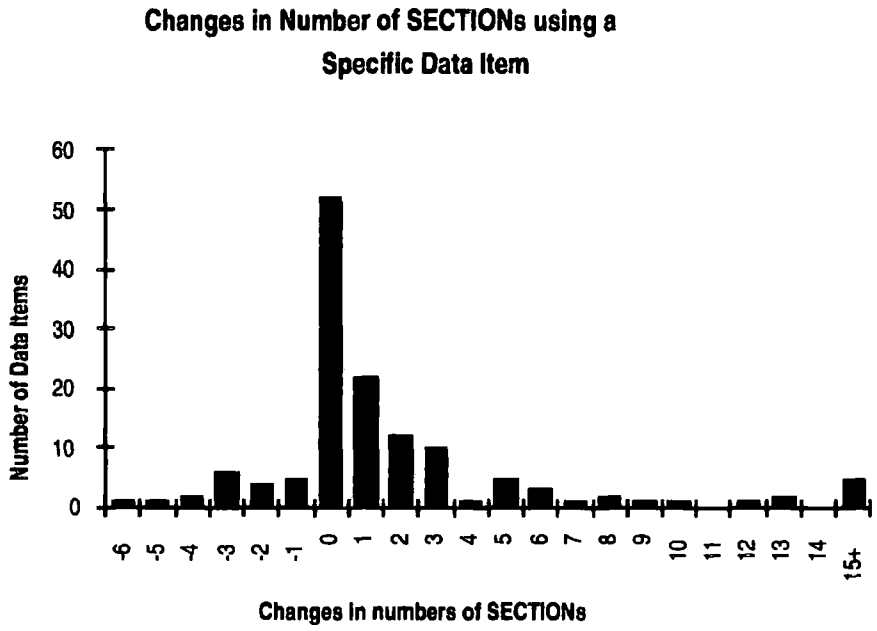


Figure 8.12: Changes in the SECTIONs' use of data items through time

In order to establish how the complexity of the data items is increasing through the evolution process, it is necessary to investigate the changes made to individual data items over time. For this study, as before, only the new addition of a data item within a SECTION or its absence is considered. It would be interesting to see if changes are also reflected within the logic of the program, but in order to gain an overview of the evolution process, a high-level understanding of the changes to the code should first be ascertained. For this reason, the study concentrates on general changes to the code rather than specific ones.

In the diagram in figure 8.13, three PERFORM graphs are shown. The edges have been hidden to increase clarity. Moving from left to right across figure 8.13, the age of the software increases. Within each of the PERFORM graphs a number of nodes are highlighted. These represent the SECTIONs within which a specific data item is used. Although slight differences in the PERFORM graphs exist between each version, these are not shown to assist comparison. The figure shows that through time the number of SECTIONs using a particular data item increases from 1 to 30. Further, the reach of data usage, in the final version, is spread across a large area of the PERFORM graph.

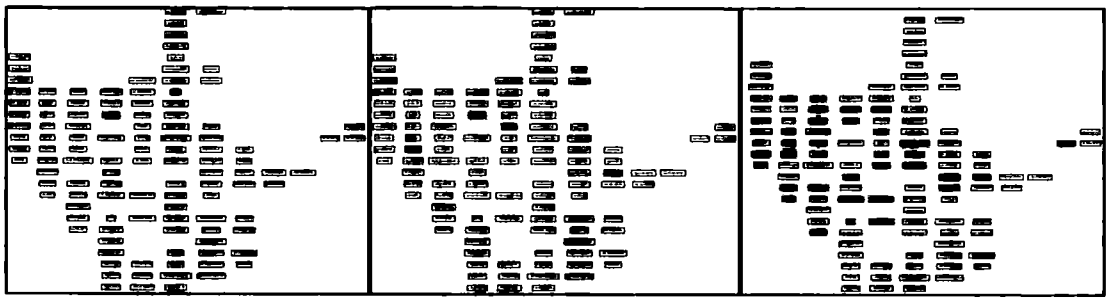


Figure 8.13: Changes in data usage over time

These results would seem to indicate that, for the majority of changes, as the evolution process progresses, so the effect of the change increases. This is in terms of the SECTIONs involved within a change. Thus, what seems fairly certain is that gaining small and manageable candidate reuse units from software becomes more difficult as the process of evolution progresses. Unless, of course, some form of preventative maintenance in carried out.

The results of the data analysis have shown that the complexity of the inter-relationship between candidates increases throughout the process of evolution. Individual data items have been shown to frequently increase in the number of SECTIONs within which they are present, the consequently the candidate reuse units become harder to separate as the evolution process progresses.

8.2.3 Summary

The results within this section have indicated how the applications within the evolution case study have evolved. The case studies have shown instances where the reachability of the data items increases in terms of the number of SECTIONs within which each data item is used. The above discussion has indicated how the extended reachability of the data items affects the resulting clusters. Thus, as the reachability of the data items increases, so the usability of the clustering approach, and in particular non-assisted approaches, as a re-modularization technique, diminishes. Therefore, while clustering techniques can produce good code re-modularizations, it is important that they are supported by other approaches such as the use of dominance tree analysis.

8.3. The Use of Graphical Representations

The final approach is to note if graphical representations of stages of the method provide information over and above that of textual representations. This section therefore evaluates if graphical representations provide essential data to the users of the method. The case study selected to assess the graphical representations takes a number of PERFORM graphs from an application and notes if

accurate predictions can be made regarding the benefits obtained from the application of the reuse reengineering method.

The method assumes that an accelerated, and perhaps more accurate, understanding of the source code can be obtained if graphical representations are used. Many of the benefits of graphical representations can only be evaluated with industrial trials of the techniques. For the purpose of this work, such a commitment on behalf of the maintenance staff to provide statistically viable results is infeasible. Therefore, other approaches to evaluating the benefit of the use of graphical representations within the method are sought.

8.3.1 The Evaluation Approach

The procedure selected is to investigate if the visual representations of the code can be used to identify where necessary reengineering effort is required. Using the full method the results of the analysis of candidate reuse units have been identified. The results of the analysis are used to form an ordering process by grouping samples as low, medium and high yields of candidate reuse units. The first point of visualisation is then carried out (the PERFORM graph of step 1) to see if any specific qualities of the graph can be used to pre-select code samples which will yield many candidate reuse units. In this way the suitability of graphical representations can be evaluated. If the initial graphical representation of the method can be used to give an indication of the likelihood of its success, then the representations can be seen to be supporting the method.

In order to establish criteria for the decision making process some qualities of the graphical representation need to be proposed. These qualities will then be identified (or their absence established) within the graph to allow decisions to be made regarding the estimated viability of the application of the method. That is, the expected yield of candidate reuse units from the application of the full method. In particular, a number of qualities are identified. These are the number of links to nodes, the number of fan-in / fan-out links per node and other visual identification cues such as the overall graph shape, its depth and its hierarchical nature.

8.3.2 Results of the Analysis of Graphical Representations

The results are shown in Figures 8.14a to 8.14c. From the application of the full method it is known that the code in Figure 8.14a results in the lowest yield of candidate reuse units whereas the code represented within 8.14c yields the greatest.

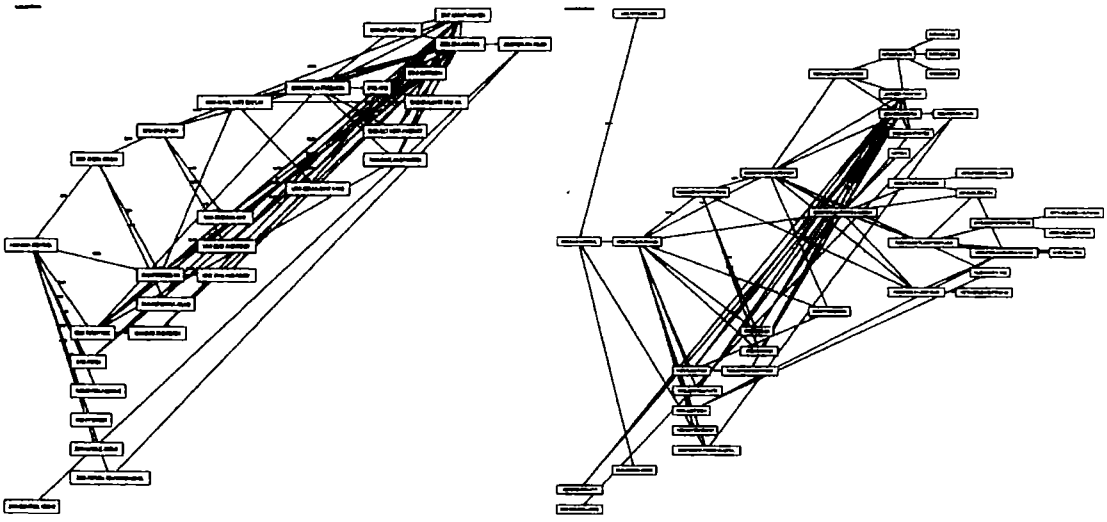


Figure 8.14a: Examples of low yield PERFORM graphs

The qualities of low yield PERFORM graphs (see Figure 8.14a) have been identified as consisting of fewer SECTIONs with these SECTIONs lacking a hierarchical structure. In addition, there are often a number of high fan-in nodes.

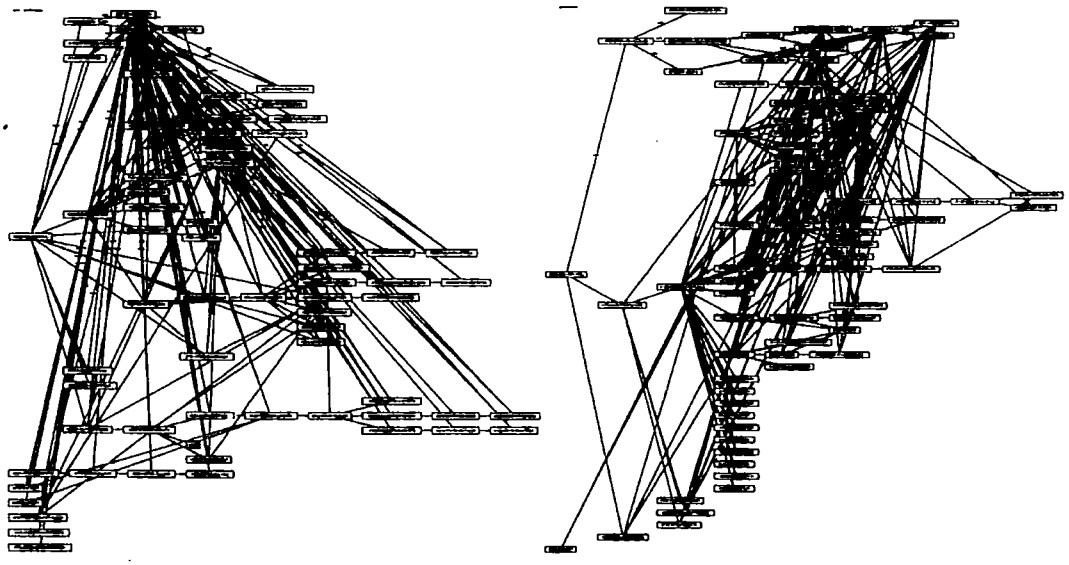


Figure 8.14b: Examples of medium yield PERFORM graphs

The qualities of the medium yield graphs (see Figure 8.14b) seem to consist of a greater number of SECTIONs but the most striking feature is their high proportion of links to nodes. In addition, the numbers of high fan-in nodes is large. This tends to increase the numbers of single SECTION candidates (service candidates) which are usually too small to be considered for reuse.

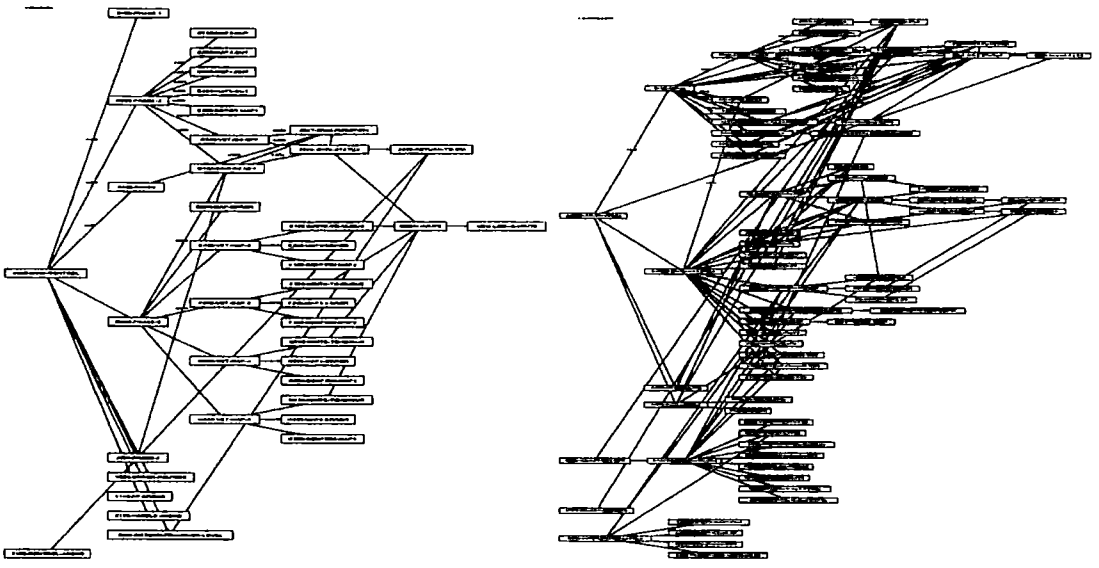


Figure 8.13c: Examples of high yield PERFORM graphs

The qualities of the high yield graph (see Figure 8.14c) are really identifiable by their hierarchical structure. Furthermore, they seem to posses fewer areas of high fan-in or fan-out.

In general, the results of the visualisation process indicates that it is feasible to broadly categorise the likely yield of candidate reuse units based on the structure of the PERFORM graph. It therefore seems that the use of graphical representations even early on within the method, has the potential to greatly enhance the method if not providing considerable benefit.

8.4. Analysis of Results

The first approach to be evaluated within this chapter was the use of the dominance tree. Results of the evolution upon dominance tree analysis have shown that candidate reuse units do tend to split into smaller candidates often due to the addition of further SECTIONs. However the studies of evolution have shown that there were minimal changes to the composition of the SECTIONs within the candidate reuse units. Further there were no migrations of SECTIONs to other candidate reuse units. The splitting of candidate re use units to a number of smaller candidates can be seen as a failing of the identification process if no additions were made to the candidate to cause the necessity for it to split. For instance, an inappropriate selection of the level of granularity may have been made. In the case studies, this was not found to be the case. In each instance where a split to a candidate reuse unit occurred, it was coupled within the addition of new SECTIONs. Thus we can assume that this factor is not a failing of the approach but rather a natural part of software evolution. It therefore concludes that the use of the dominance tree derives 'good' candidate reuse units in that they appear to require little re-work to support the process of evolution.

The second approach evaluated within this chapter was the use of data clustering. The results of the study on a non-assisted data clustering approach indicated that the candidate reuse units identified were very small. The size of the candidates, many composed of around 3 SECTIONs, is probably too small to be worthy of reuse. The grouping of the candidates to form larger ones may be considered but in this case it is necessary to perform the activity with assistance from other sources such as the results of the dominance tree analysis used for the assisted approach.

The assisted approach identified large candidate reuse units that were based on the results of the dominance tree analysis. The results indicated that, coupled within minor changes in the structure of these candidates, from the results of the cluster analysis, clear interfaces between the data items of the candidate reuse units were formed. However, the effect of evolution tended to reduce the clarity of the interface.

The changes in data evolution are coupled with changes to the structure of candidate reuse units due to the splitting process. Thus, with the assisted process applying the comparison of both evolutionary changes (data and PERFORM structure), it is possible for one or both sets of changes to be responsible for the reduced simplicity of the data sharing interfaces between the candidate reuse units. In order to reduce the effect of the PERFORM structure changes, candidates were grouped as near as possible to their original composition (i.e. before the evolution of the candidate reuse units occurred). The results of the process showed a reduction in the complexity of the interfaces. This is to be expected, as there are fewer candidate reuse units, the number of interactions will be reduced. Regardless of this, the simplicity of the interfaces prior to evolution was not re-established. Thus, it is necessary to conclude that the process of evolution does have a detrimental effect on the data.

This consequence of evolution was shown in Section 8.2.2 by means of the changes in reachability of evolution. The fact that data items become gradually de-localised across the code does mean that the process of data clustering will be less successful over time. This seems to support the assumption within the reuse reengineering method that data clustering should be a secondary task towards candidate reuse unit identification, and therefore an assisted approach is more suited to legacy systems.

The third and final approach to be evaluated within this chapter was the use of graphical representations to support the steps of the reuse reengineering method. A number of different approaches could be used to test the suitability of graphical representations. The procedure taken, however, was to investigate if the overall suitability of the application of the method on a specific piece of software could be gained from the first visualisation point within the method. The results within Section 8.3.2 show that it is possible to broadly categorise the PERFORM graphs into low, medium and high yield code samples. Consequently it is feasible to make predictions based on the graphical representations, The provision of such information will aid the enthusiasm of the users of the method (since the likelihood of gaining successful results is known early on), the use of graphical representations is more than justified. Additionally, the possibility that the graphical representation can

give an indication of code samples which are unsuited to the application of this method (for instance, those code samples which only contain a single functionality) means they form a vital and integral part of the overall method.

Chapter 9. Conclusions & Further Work

This chapter summarizes the research carried out in this thesis, and suggests possible directions and ideas for future research related to this work.

9.1. Objective of this Work

The objective for this work has been to define an approach for the identification, extraction and encapsulation of potential reuse candidates from legacy systems. A method has been devised whereby legacy systems are analysed to investigate whether they may be re-modularised into numerous smaller candidate reuse units. These new candidate reuse units can later be evaluated for their potential as reuse candidates. The application of a number of case studies has found that through the use of the method, the re-modularization of legacy systems is feasible. Furthermore, later studies of the effects of the evolution process on these candidate reuse units has shown that there are benefits of the method's application.

9.2. Review of Criteria for Success

In Chapter 1 a number of criteria for success were defined against which this work could be judged. These criteria will now be reconsidered to investigate the successes and failures of the approaches taken.

- Describe a method which will effectively collate existing techniques into a number of easy to follow steps.

A 10 step method has been defined within this thesis. Furthermore, some steps may consist of a number of tasks. The stepwise approach of the method assists users in its execution. Objectives and expected outcomes of each of the steps have also been defined. In addition, small examples are

provided for the steps that assist users in gaining an understanding of how the steps can be applied to their software applications. However, currently, the method has only been applied to software applications by those involved with the design of the method, currently it is not possible to indicate the usability of the method with respect to non-experienced users.

- Add to, or improve upon, the existing techniques where it proves necessary to support the reuse candidate identification process.

The dominance tree analysis for candidate identification has been selected from the RE^2 programme (described in Chapter 3). This is also supported by the use of clustering techniques. Both of these approaches are assisted considerably by the use of graphical representations (see Chapter 8) and simplification procedures (Step 7).

The method includes a number of simplification techniques to improve the 'readability' of the visual representations. One of the most significant contributions of the method is the realisation of the significance of non-functional requirements. The results of the case study showed that reductions of 50% of the information being represented could be made by the temporary excluding non-functional requirements.

- Provide support for both top-down and bottom-up comprehension processes.

In addition to the other supporting techniques described above, another distinguishing aspect of the approach is the inclusion of users and designers within the normal execution of the method (Step 6). The involvement of designers within the method allows support for both top-down and bottom-up comprehension strategies. This is an unusual provision within reuse reengineering as most existing methods support only one approach. Thus, the support of both strategies is an important contribution to reuse reengineering methods through assisting program comprehension.

- Provide consistent and complete support for a single language which has been frequently used to implement today's legacy systems.

The reuse reengineering method has concentrated on the application of the COBOL language. This language has been selected since it is estimated that there are over 150 billion lines of COBOL which is two and a half times more than is written within C [Hoffnagl96]. The method supports most aspects of the language, but if GO TO statements are present these must first be removed. Further enhancements to the method would be feasible if more detailed automated support i.e. for control flow was available. This would assist the analysis of the later stages of the method. Other languages can be analysed using

the method, but, in some cases, slight modifications would be necessary. For instance, modifications will be necessary if C is to be analysed in order for the method to be able to cope with the use of pointers.

- Use graphical displays of information to support the decision making process necessary for the use of the method.

Within Chapter 8 the use of graphical representations was evaluated. Their use has been shown to have many advantages, but their greatest benefit is in terms of gaining an immediate overview of the software. Chapter 8 sets the hypothesis that if graphical representations can indicate the need to apply the reengineering approach, then the use of graphical representations is well justified. The results of the evaluation do seem to justify their use.

The issues of scale are however important. As larger software applications are analysed, so the complexity of the graphical representation increases. While the abstractness of the graphical representations does reduce complexity in comparison to the purely textual representations, eventually a threshold is reached beyond which the graphical representation cannot be considered as understandable. However, the use of graphical representations along with simplification strategies does appear to reduce this problem.

- Evaluate the results of the use of the method against a number of industrial sized applications.

A number of case studies have been carried out. These represent 'real' applications that are still in commercial use. Each application has been highly maintained. In many cases the code modules will be nearly 20 years old. In total, over 1/3 of a million lines of COBOL have been analysed within the case studies of this thesis. The largest code module analysed is over 40,000 lines of code.

Within the case studies, applications of three vastly different sizes are analysed ranging from 400 to 40,000 lines of code. The results obtained show that the benefits of the application of the method increased with the size of the code. For instance, greater numbers of and larger reuse candidates were identified from the bigger samples. In general it was found that the method scales well to support large applications.

- Test the method to see if it is supportive of the process of software evolution

Chapter 8 has described how software applications change due to the process of software evolution. This knowledge is used to evaluate the suitability of the method. The demonstration that the method does not have an adverse effect on the software based upon historical changes made to the application, but is instead beneficial to it, would seem to be a good predictor of the future potential of the reuse reengineering method. The results have shown that the dominance tree is a good mechanism to support evolution but not the non-assisted strategy of data clustering. Overall, the results seem to indicate that the combined dominance tree and assisted data clustering approaches provide better overall 'reuse candidates' for evolution.

- Indicate the type of tools which should be available to support the use of the method.

For each stage of the method recommendations for tool support have been indicated within Chapter 4. The method does have some support tools, for instance, for the generation of the PERFORM graphs and the dominance trees. In addition, existing graphical display tools have been used. However, additional tools would be required for commercial use of the method. In particular, access is required to a detailed COBOL analyser that would assist the generation of the data interactions (Steps 4 and 5). Tools that are able to carry out this function are available, but are generally very expensive. Further work would need to be carried out to investigate the requirements of a slicing tool that will allow the splitting of functions with more than one functionality (for support of Step 10). In addition, improvement to existing graphical display tools are required, in particular, involving automated layout facilities.

9.3. Recommendations on Using the Method

When identifying candidates for reengineering it is important to take account of two types of decision making factors. Firstly, the current and future business decisions of the company embarking on reengineering must be evaluated. Secondly the features of the resulting dominance tree should be reviewed in order that appropriate decisions regarding the suitability of the module to reengineering are assessed.

These two factors are considered further below.

9.3.1 Business Decisions

In order that reengineering is justified it is necessary to take account of the business decisions made and how these effect the company's software. Thus when proposing reengineering:

1. The module to be reengineered should be important to the business i.e. represent crucial business functionality
2. The module should show constantly increasing maintenance costs, probably because of the process of continued maintenance

9.3.2 Feature of PERFORM Graphs

To identify potentially suitable modules the graphical representations of the PERFORM graphs can be used to select appropriate code and to prevent unnecessary analysis of code samples which will not benefit from the approach. The characteristics identifiable of a PERFORM graph are:

1. A low yield of reuse candidates is identifiable by graph which contain a relatively small number of nodes i.e. approximately 20 or 30.
2. A low yield of reuse candidates is identifiable by a graph which can be seen to lack a hierarchical structure
3. A low yield of reuse candidates is identifiable by a graph which can be seen to have a high proportion of fan-in nodes.
4. A potential high yield of reuse candidates is the opposite of points 1 – 3 i.e. containing a high number of nodes, a hierarchical structure and a low proportion of fan-in nodes.

9.3.3 Features of the Dominance Tree

A candidate that is identified from the dominance tree, if it suitable for selection as a reuse candidate, should have the following features:

1. At least two potential candidates should be identified within a COBOL module (obviously splitting functionality is not necessary if only a single functionality is identified within a module) if the reengineering process is to be considered.
2. The module identified should be composed of five or more SECTIONs.
3. The depth of the PERFORM structure should be equal to or greater than two PERFORMs
4. The proportion of directly dominated notes should be low, preferably less than three in total. Ideally candidates should have no directly dominated nodes.
5. The functionality of the candidate should be identified as responsible for an important business function
6. Ideally the identified business function should be one that will remain within use within the company for some time or have been identified as one which is constantly in need of modification

9.4. The Thesis Contribution

In summary, the major contributions of this work are in the provision of:

- a 10 Step method for reuse reengineering and the addition of other supporting tasks for the more complex steps.
- a number of strategies for graph simplification including those relating to the implementation of non-functional requirements.
- strategies for data interface simplification and approaches to deal with grouping of related data items.
- new graphical representations for COBOL REDEFINES and consists relationships.
- the combination of top-down and bottom-up support and the inclusion of an 'expert' based concept assignment programme.
- an improved understanding of how software applications evolve.

9.5. Directions for Further Research

A number of research directions are possible from this work. Some of them are listed in the section below. In order to give some indication of the timescales required for their implementation, each is categorised into either short, medium or long term goals.

Short Term

- slicing of SECTIONs offering more than one functionality to increase the level of granularity resulting from the automated approach.
- strategies for management of the reuse program to ensure the appropriate support mechanisms are in place for the replacement of the legacy code with the new reuse units.

Medium Term

- metrics to define when an application requires re-modularization.
- mechanisms to 'undo' the effects of increasing change reachability.

Long Term

- automation of the reuse reengineering process so that the software applications automatically evolve.
- metrics for the prediction of evolutionary trends.

This thesis has contributed an approach to the improved maintainability of software and understanding of the process of software evolution.

References

Achee94 — Achee B.L., Carver D.L., 'A Greedy Approach to Object Identification in Imperative Code' Workshop on Program Comprehension, IEEE Press, 1994

Agrest88 — Agresti W.W., 'The Minnowbrook Workshop on Software Reuse: a summary report' in Tracz W., (ed.), 'Software Reuse: Emerging Technology', IEEE Computer Press, 1988

Ahrens95 — Ahrens J.D., Prywes N.S., 'Transition to a Legacy and Reuse Based Software Lifecycle', IEEE Computer, 1995

ANSI83 — 'IEEE Standard Glossary of Software Engineering Terminology', IEEE Press, 1983

Ball94 — Ball T., Eick S.G., 'Software Visualisation in the Large', in Proceedings of the 1994 IEEE Symposium on Visual Languages', IEEE Press, 1994

Barns91 — Barns B.H., Bollinger T.B., 'Making Reuse Cost Effective', IEEE Software, Vol. 8, No. 1, January 1991

Basili90 — Basili V.R., Caldiera G., 'Reusing Existing Software' in International Conference on Software Engineering (ICSE), Nice 1990

Bassett97 — Bassett P.G., 'Framing Software Reuse: lessons from the real world', Yourdon Press, 1997

Beck89 — Beck, K., Cunningham W., 'A Laboratory for Teaching Object Oriented Thinking', Proceedings of the ACM OOPSLA '89 Conference, 1989

Biggerst89a — Biggerstaff T., Richer C., 'Reusability Framework Assessment, and Directions', in Biggerstaff T., Perlis A., 'Software Reusability: Concepts and Models', ACM Press, Vol. 1, 1989

Biggerst89b — Biggerstaff T.J., 'Design Recovery for Maintenance and Reuse', IEEE Software, 22(7), July 1989

Biggerst94 Biggerstaff T.J., Mitbander B.G., Webster D.E., 'Program Understanding and the Concept Assignment Problem', Communication of the ACM, May 1994

Binder96 Binder R.V., 'Modal Testing Strategies for OO Software', IEEE Computer, November 1996

Blum96 Blum B.I., 'Beyond Programming: to a new era of design', Oxford, 1996

Bodhuin94 Bodhuin T.M., 'An Interaction Paradigm for Impact Analysis', M.Sc. Thesis, University of Durham, 1994

Boehm75 Boehm B.W., 'The High Cost of Software', in Horowitz E., Practical Strategies For Developing Large Software Systems, Addison Wesley, 1975

Boldyref96 Boldyreff C., Burd E.L., Hather R.M., Munro M., Younger E.J., 'Greater Understanding Through Maintainer Driven Traceability', in Proceedings of the International Workshop in Program Comprehension, Germany, IEEE Press, March 1996

Booch91 Booch G., 'Object Oriented Design with Applications', Benjamin Cummings, 1991

Brooks83 Brooks R., 'Towards a Theory of the Comprehension of Computer Programs', International Journal of Man-Machine Studies, Vol. 18, No. 6, 1983

Brooks87 Brooks F.P., (Jr.) 'No Silver Bullet: Essence and accidents of software engineering', IEEE Computer, No 20, 1987

Brown88 Brown M.H., 'Algorithm Animation', in ACM Distinguished Dissertations, MIT Press, New York, 1988

Brown93 Brown, M.H., Najork M.A., 'Algorithm Animation Using 3D Interactive Graphics', DIG90, September 1993

Bruegge93 Bruegge B., Gottschaalk T., Luo B., 'A Framework for Dynamic Program Analysers', SIGPLAN Notices: Proceedings of the 8th Annual ACM Conference on Object Oriented Programming Systems, Languages and Applications, Sept. 1993

Caldiera91 Caldiera G., Basili V., 'Identifying and Qualifying Reusable Software Components', IEEE Computer, Feb 1991

Canfora94 Canfora G., Cimitile A., Tortorella M., Munro M., 'A Precise Method for Identifying Reusable Abstract Data Types in Code', International Conference on Software Maintenance - ICSM'94, IEEE Press, 1994

Canfora95 Canfora G., Cimitile A., Visaggio G., 'Assessing Modularization and Code Scavenging Techniques', Journal of Software Maintenance, Vol. 7, No. 5, October 1995

Cardenas91 Cardenas-Garcia S., Zelkolwitz V., 'A Management Tool for Evaluation of Software Designs', IEEE Transactions on Software Engineering, Sept. 1991

Cheatham84 Cheatham T.E., (Jr.), 'Reusability Through Program Transformations', IEEE Transactions on Software Engineering, Vol. 10, no 5, Sept 1984

Chikofsk90 Chikofsky E., Cross J., 'Reverse Engineering and Design Recovery - a taxonomy', IEEE Software, 1990

Cimitile95 Cimitile A., Visaggio G., 'Software salvaging and the call dominance tree', The Journal of Systems and Software, vol. 28, No. 2, February 1995

Cimitile98 Cimitile A., De Lucia A., Di Lucca G.A., 'An Experiment in Identifying Persistent Objects in Large Systems', International Conference on Software Maintenance, IEEE Press, 1998

Coad92 Coad P., 'Object-oriented Patterns', Communications of the ACM, Vol. 33, No. 9, 1992

Colbrook90 Colbrook A., Smythe C., Darlison A., 'Data Abstraction in a Software Reengineering Reference Model', Proceedings of the IEEE Conference on Software Maintenance, IEEE Computer Society Press, 1990

Coleman92 Coleman D., Hayes F., Bear S., 'Introducing Objectcharts or How to Use Statecharts in Object-oriented Design', IEEE Transactions on Software Engineering, 18(1), January 1992

Conn88 Conn R., 'The Ada Software Repository and Software Reusability', In Tracz W (ed), 'Software Reuse: Emerging technology', IEEE Computer press, 1988

Consens92 Consens M., Mendelzon A., Ryman A., 'Visualising and Querying Software Structures', 14th International Conference on Software Engineering, ICSE'92. May 11-15, 1992

Cramer91 Cramer J., Hünnekens H., Schäfer W., Wolf S., 'The MERLIN approach to the Re-use of Software Components', in Dusink E., Hall P.A.V., (eds.) 'Software Re-use Utrecht 89', Chapter 6, Springer-Verlag, 1991

Cutillo93 Cutillo F., Fiore P., Visaggio G., 'Identification and Extraction of Domain Independent Components in Large Programs', Working Conference on Reverse Engineering, WCRE'93, IEEE Press, 1993.

DeBaud97 DeBaud J., 'DARE: Domain-Augmented Reengineering', Working Conference on Reverse Engineering, IEEE Press, 1997

DeLucia95 De Lucia A., 'Identifying reusable Functions in Code Using Specification Driven techniques', M.Sc. thesis, University of Durham, 1995

DeLucia96 De Lucia A., Fasolino A.R., Munro M., 'Understanding Function Behaviors through Program Slicing', International Workshop on Program Comprehension, IEEE Press, 1996

Deursen98 van Deursen A., Moonen L., 'Type Interence of COBOL Systems', Fifth Working Conference on Reverse Engineering, IEEE Press, 1998

Dineur89 Dineur A., Picard P., 'SFINX: Tool Integration on a PCTE Based Software Factory, in Madhavji H.N., Schäfer W., Weber H., (eds.), Proceedings of the 1st International Conference on Software Development Environments and Factories, Berlin 1989, Pitman Publishing London 1990

Dubinsky89 Dubinsky E., Freudenberger S., Schonberg E., Schwartz J.T., 'Reusability of design for large software systems: an experiment with SETL Optimizer', in Biggerstaff T, Perlis A, Software Reusability: Concepts and Models, ACM Press, Vol. 1, 1989

Edwards93 Edwards H., Munro M., 'RECAST: Reverse Engineering from COBOL to SSADM Specifications', International Conference on Software Engineering, IEEE Press, 1993

Edwards96 Edwards H., Munro M., 'West R., The RECAST Method for Reverse Engineering', Blackwell, 1996

Eick96 Eick S.G., Ward A., 'An Interactive Visualisation for Message Sequence Charts', International Conference on Program Comprehension; IWPC'96, IEEE Press, 1996

Everitt93 Everitt B.S., 'Cluster Analysis', Edward Arnold, 1993

Fairchil88 Fairchild K.M., Poltrock S.E., Furnas G.W., 'Semnet: Three-dimensional Graphics Representations of Large Knowledge Bases', Cognitive Science and Its Applications for Human Computer Interaction, Lawrence Erlbaum Associates, 1988

Fowler97 Fowler M., 'Analysis Patterns: Reusable Object Models', Addison Wesley, 1997

Frakes88 Frakes W.B., Nejmeh, 'An Information System for Software Reuse' Tracz W (ed.), 'Software Reuse: Emerging Technology', IEEE Computer press, 1988

Freeman84 Freeman P., Reusable Software Engineering: Concepts and Research Directions, Proceedings of the ITT Workshop on Reusability Through Program Transformation on Software Engineering, Vol. 10, No. 5, Sept. 1984

Freeman87 Freeman P., 'A Conceptual analysis of the Draco approach to constructing software systems', IEEE Transactions on Software Engineering, Vol. 13, July 1987

Gaffney88 Gaffney J.E., Durek T., 'Software Reuse - Key to Enhanced Productivity: some qualitative Models', Software Productivity Consortium, SPC-TR-88-015, 1988

Gall92 Gall H., Klösch R., 'Reuse Engineering: software construction from reusable components', IEEE Computer Software and Applications Conference COMPSAC'92, IEEE Press, 1992

Gall94 Gall H., Klösch R., 'Managing Uncertainty in a Object Recovery Process', 5th International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems (IPMR'94), July 1994

Gall95 Gall H., Klösch R., 'Finding Objects in Procedural Programs: an alternative approach', Working Conference on Reverse Engineering, WCRE95, IEEE Press, 1995

Gamma95 Gamma E., Helm R., Johnson R., Vlissides J., 'Design Patterns: elements of reusable object-oriented software', Addison Wesley, 1995

Goguen84 Goguen J.A., 'Parameterized Programming', IEEE Transactions in Software Engineering, vol. 10, No. 5, Sept 1984

Glaser90 Glaser, E.L., Pyle I.C., (Eds.), 'Dictionary of Computing', Oxford, 1990

Griswold93 Griswold W.G., Notkin D., 'Automated Assistance For Program Restructuring', ACM Transactions on Software Engineering and Methodologies', Vol. 2, No. 3., July 1993

GV3D95 http://www.omg.unb.ca/hci/projects/hci-gv3D.html

Hall88 Hall P.A.V., 'Software Components and Reuse', Software Engineering Journal, Sept. 1988

Harman97 Harman M., Danicic S., 'Amorphous Program Slicing', International Workshop on Program Comprehension', IEEE Press, 1997.

Harris95 Harris D.R., Reubenstein H.B., Yeh A.S., 'Reverse Engineering to the Architectural Level', International Conference on Software Engineering, ICSE'95, ACM Press, 1995

Hecht77 Hecht M.S., 'Flow Analysis of Computer Programs', Elsevier, 1977

Hendley95 Hendley R.J., Wood A.M., Beale R., Drew N.S., 'Narcisses: Visualising Information', University of Birmingham, 1995

Hoffnagl96 Hoffnagle G., 'Systems and Software Evolution: Practical Aspects and Cautionary Tales', Keynote Address, 4th International Workshop on Program Comprehension, Berlin, IEEE Press, March 1996

Jackson94 Jackson D., Rollins E.J., 'Abstracting Mechanisms for Pictorial Slicing', International Workshop on Program Comprehension, IEEE Press, 1994

Jacobson91 Jacobson I., Lindström F., 'Reengineering of Old Systems to an Object-oriented architecture', OOPSLA, ACM Press, 1991

Jones84 Jones T.C., 'Reusability in Programming: a survey of the state of the Art', IEEE Transactions in Software Engineering, Sept. 1984

Kaiser87 Kaiser G.E., 'Melding Software Systems from Reusable Building Blocks', IEEE Software, Vol. 4, No. 4, July 1987

Kang88 Kang K.C., 'A Reuse Based Software Development Methodology', in Tracz W., (ed) 'Software Reuse Emerging Technology', IEEE Computer Press, 1988

Koike95 Koike Labs., 'VisuaLinda: 3D Visualisation of Parallel Linda Programs', Journal of Parallel and Distributed Computing. 18, 1993

Korel97 Korel B., Rilling J., 'Dynamic Program Slicing in Understanding of Program Execution', International Workshop on Program Comprehension, IEEE Press, 1997

Korel98 Korel B., Rilling J., 'Program Slicing in Understanding of Large Programs', International Workshop on Program Comprehension, IEEE Press, 1998

Krueger92 Krueger C.W., 'Software Reuse', ACM Computing Surveys, Vol. 24, No. 2, June 1992.

Langerga84 Langergan R.G., Grasso C.A., 'Software Engineering with Reusable Designs and Code', IEEE Transactions on Software Engineering, SE10, 1984

Lano91 Lano K., 'Z++ an Object Oriented extension to Z', Proceedings of the 5th Annual Z User Meeting, Springer Verlag, 1991

Lano93 Lano K., Breuer P.T., Haughton H., 'Reverse engineering COBOL via Formal Methods', Software Maintenance Research and Practice, Vol. 5, 1993

Lano94 Lano K., Haughton H., 'Reverse Engineering and Software Maintenance: a practical approach', MCGraw-Hill, 1994

Latour88 Latour L., 'SEE: An Automated Tool to Facilitate Reuse in Ada Project Development' in Tracz W., (ed), 'Software Reuse: Emerging technology', IEEE Computer press, 1988

Lehman97 Lehman M.M., Ramil J.F., Wernick P.D., Perry D.E., 'Metrics and Laws of Software Evolution - the nineties view', Symposium on Software Metrics, IEEE Press, Nov 1997

Lejter92 Lejter D.C., Meyers S Reiss S., 'Support for Maintaining Object-oriented Programs', IEEE Transactions on Software Engineering, Vol. 18, No. 12, December 1992

Lenz87 Lenz M., Schmid H.A,. Wolf P., 'Software Reuse Through Building Blocks', IEEE Software, July 1987

Letovsky86 Letovsky, S., and Soloway, E. 'De-localized Plans and Program Comprehension', IEEE Software. May 1986, Vol. 19, No. 3.

Lientz80 Leintz B.P., Swanson E.B., 'Software Maintenance Management', Addison Wesley, 1980

Littman86 Littman, D.C., Pinto, J., Letovsky, S., and Soloway, E., 'Mental Models and Software Maintenance. Empirical Studies of Programmers', Albex, Norwood NJ, 1986.

Liu90 Liu S., Wilde N., 'Identifying Objects Conventional Procedural Language an Example of Data Recovery', International Conference on Software Maintenance, IEEE Press, 1990

Livadas94 Livadas P.E., Johnson T., 'A New Approaches to Finding Objects in Programs', Journal of Software Maintenance: Research and Practice, Vol. 6, 1994

Lubars88 Lubars M.D., 'Code Reusability in the Large versus Code Reusability in the Small', in Tracz W (ed.), 'Software Reuse: Emerging technology', IEEE Computer Press, 1988

McNamara84 McNamara B., 'Japanese Software Factories' in Standish TA, 'An Essay on Software Reuse', IEEE Transactions on Software Engineering, Vol. 10, No. 5, Sept. 1984

Meekel88 Meekel J., Viala M., 'Logiscope: A tools for maintenance', Proceedings of the International Conference on Software Maintenance, ICSM'88, IEEE Press, 1988

Meyer87 Meyer B., 'Reusability: the case for Object-oriented Design', IEEE Software, Vol. 4, No. 2, March 1987

Moher88 Moher B.A., 'PROVIDE: A Process Visualization and Debugging Environment', IEEE Transactions on Software Engineering Vol. 14, No. 6, June 1988

Montes98 Montes de Oca C., Carver D.L., 'Identification of Data Cohesive Subsystems Using Data Mining Techniques', International Conference on Software Maintenance, IEEE Press, 1998

Morrison87 Morrison R., Brown A.L., Carrick R., Connor R.C.H., Dearle A., Atkinson M.P., 'Polymorphism, persistence and software re-use in a strongly typed object-oriented environment', Software Engineering Journal, Vol. 2, No. 6, Nov 1987

Myers85 Myers W., 'An Assessment of the Competitiveness of the US Software Industry', IEEE Computer Society NY, March 1985

Myers90 Myers B.A., 'Taxonomy of Visual Programming and Program Visualisation', Journal of Visual Languages and Computing, (1), 1990

Ning89 Ning J.Q., 'A knowledge Based Approach to Automatic Program Analysis', Ph.D Thesis, University of Illinois at Urbana-Campaign, October 1989

Ning93 Ning J.Q., Engberts A., Kozaczynski W., 'Recovering Reusable Components from Legacy Systems by Program Segmentation', Working Conference on Reverse Engineering, IEEE Press, 1993

Ning96 Ning J.Q., 'A Component-based Software Development Model', International Conference on Software Engineering, ICSE96, IEEE Press, 1996

Parikh83 Parikh G., Zvegintzov N., 'Tutorial on Software Maintenance', IEEE Computer Society Press, Silver Spring Maryland, 1993

Pree95 Pree W., 'Design Patterns and Object-oriented Software Development', Addison Wesley, 1995

Price93 Price, B.A., Baeker R.M., Small I.S., 'A Principled Taxonomy of Software Visualisation', Journal of Visual Languages and Computing. No. 4, 1993.

Prieto-D87a Prieto-Diaz R., Neighbors J.M., 'Module Interconnection Languages', Journal of Systems and Software, Vol. 6, 1987

Prieto-D87b Prieto-Diaz R., Freeman P., 'Classifying Software for Reusability', IEEE Software, 1987

Purtilo91 Purtilo, J.M., Atlee, J.M., 'Module reuse by interface adaptation', Software - Practice and Experience, vol. 21, no 6, June 1991

PVMTrace95 The PVMTrace Project. On-line Thesis Proposal, University of New Brunswick, 1995

Ranganat91 Ranganathan S.R., 'Prolegomena to Library Classification', in Cramer J, Hünnekens H., Schäfer W., Wolf S., 'The MERLIN approach to the Re-use of Software Components', In Dusink E., Hall P.A.V., (Eds.) 'Software Re-use Utrecht 89', Chapter 6, Springer-Verlag, 1991

Reasonin http://www.reasoning.com/

Reiss85 Reiss S.P., 'PECAN: Program Development Systems that Support Multiple Views', IEEE Transactions on Software Engineering, SE11 (3), March 1985

Reiss90 Reiss S.P., 'Interacting with the FIELD Environment', Software-Practice and Experience, Vol. 20 No. 1, June 1990

Reiss93 Reiss S.P., 'A Framework for Abstract 3D Visualisation', Proceedings of the 1993 IEEE Symposium on Visual Languages, August 1993

Reps84 Reps T., Teitlbaum T., 'The Synthesisor Generator' SIGPLAN Notices, Vol. 19, No. 5, 1984

Reubenst91 Reubenstein, H.B., Waters, R.C, 'The Requirements Apprentice: automated assistance for requirements acquisition', IEEE Transactions on Software Engineering, Vol. 17, No. 3, March 1991

Rich90 Rich C., Waters R.C., 'The Programmer's Apprentice', ACM Frontier Series, ACM Press, 1990

Roman93 Roman G., Cox K.C., 'A Taxonomy of Visualisation Systems', IEEE Computer, December 1993

Rugaber90 Rugaber S., Ornburn S.B., LeBlanc R.J., (Jr.) 'Recognising Design Decisions in Programs', IEEE Software (7), January 1990

Rumbaugh91 Rumbaugh J., Blaha M., Premerlani W., Eddy R., Lorensen W., 'Object-oriented Modelling and Design', Prentice Hall, 1991

Shneider79 Shneiderman B., and Mayer R., 'Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results', International Journal of Computer and Information Sciences. 1979, Vol. 8, No. 3

Siff97 Siff M., Reps T., 'Identifying Modules via Concept Analysis', Internation Conference on Software Maintenance, IEEE Press, 1997

Sneed92 Sneed H.M., 'Migration of Procedurally Oriented COBOL programs in an Object-oriented Architecture', International Conference on Software Maintenance '92, IEEE Press, 1992

Sneed95 Sneed H.M., Nyáry E., 'Extracting Object-oriented Specifications from Procedurally Oriented Programs', Working Conference on Reverse Engineering WCRE'95, IEEE Press, 1995

Sneed96 Sneed H.M., 'Object-oriented COBOL Recycling', Working Conference on Reverse Engineering WCRE'96, IEEE Press, 1996

Snyder86 Snyder A., 'Encapsulation and inheritance in object-oriented programming languages', SIGPLAN Notices OOPSLA '86. Object-Orientated Programming Systems, Languages and Applications, Nov 1986

Soloway84 Soloway E., and Ehrlich, K., 'Empirical Studies of Programming Knowledge', IEEE Transactions on Software Engineering', Vol. SE-10, No. 5, September 1984

Soloway86 Soloway E., 'Learning to Program = Learning to Construct Mechanism and Explanations', Communication of the ACM, June 1986, Vol. 29, No., 6

Spicer91 Spicer K.L, Umphress D.A, 'A method for mapping an analysis to a reusable design', Ada letters, Vol. 11, No. 9, Nov-Dec 1991

Standish84 StandishT.A., 'An Essay on Software Reuse', IEEE Transactions on Software Engineering, Vol. 10, No. 5, September 1984

Steckel92 Steckel M., Brade K., Guzdial M., Soloway E., 'Whorf: A visualisation tool for softwarre maintenance', Proceedings 1992 IEEE Workshop on Visual Languages', IEEE Press, 1992

Storey95 Storey M-A.D., Muller H.A., 'Manipulating and Documenting Software Structures Using SHriMP Views', Proceedings of the International Conference on Software Maintenance ICSM'95, 1995

Tortorel94 Tortorella M.E., 'Identification of Abstract Data Types in Code', M.Sc. Thesis, University of Durham, 1994

Tracz87 Tracz W., 'Ada reusability efforts: a survey of the state of the practice' Proceedings of the fifth National Conference on Ada technology, Arlingtion Va., March 1987

Troy81 Troy D.A., Zweden S.H., 'Measuring the Quality of Structured Designs', Journal of Systems and Software, Vol. 2 1981

VanZuyle93 VanZuylen H.J., 'The REDO Compendium: reverse engineering for software maintenance', Wiley, 1993

Venkates91 Venkatesh G.A., 'The semantic approach to program slicing', ACM SIGPLAN Notices, Vol. 26, No 6., 1991

Vlisside96 Vlissides J.M., Coplin J., Kerth N.L., 'Pattern Languages of Program Design: 2', Addison Wesley, 1996

Ward94a Ward M., 'Language Oriented Programming', Software - Concepts and Tools, No. 15, 1994

Ward94b Ward M., 'Reverse Engineering through Formal Transformations', The Computer Journal, Vol. 37, No. 9, 1994

Ward95 Ward M., Bennett K.H., 'Formal Methods for Legacy Systems', Journal of Software Maintenance: research and practice, Vol. 7, No. 3, May/June 1995

Ware93 Ware C., Hui D., Franck G., 'Visualizing Object Oriented Software in Three Dimensions', Proceedings of CASCON '93, October 1993

Weiser84 Weiser M., 'Program Slicing', IEEE Transactions on Software Engineering, Vol. 10, No 4., July 1984

Wiedenbe86 Wiedenbeck S., 'Processes in Computer Program Comprehension. Empirical Studies of Programmers', Albex, Norwood NJ, 1986.

Wiedenbe91 Wiedenbeck S., 'The Initial Stage of Program Comprehension', International Journal of Man-Machine Studies, Vol. 35, No. 4, October 1991

Wilkerso90 Wilkerson B., 'How to Design an Object Based Application', Develop, April 1990

Wong95 Wong K., Tilley S.R., Muller H.A., Storey M.D., 'Structural Re-documentation', IEEE Software, January 1995

Wood89 Wood M., 'Software Function Frames: an approach to the classification of reusable software through conceptual dependency', Ph.D. Thesis, University of Strathclyde, January 1989

Woodfield87 Woodfield S.M., Embley D.W., Scott D.T., 'Can Programmers Reuse Software' IEEE Software, July 1987

Yang97 Yang H., Luker., Chu W.C., 'Code Understanding through Program Transformation for Reusable Component Identification', International Workshop on Program Comprehension, IEEE Press, 1997

Yeh95 Yeh A.S., Harris D.R., Reubenstein H.B., 'Recovering Abstract Data Types and Object Instances from a Conventional Procedural Language', Working Conference on Software Maintenance; WCRE'95, IEEE Press, 1995