Durham University

Durham E-Theses

Identifying reusable abstract data types in code

Maria Tortorella

How to cite:

Tortorella, Maria (1994) Identifying reusable abstract data types in code. Masters thesis, Durham University.

Use policy

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a https://etheses.durham.ac.uk/id/eprint/5092/ is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the full Durham E-Theses policy for further details.

Academic Support Office, The Palatine Centre, Durham University, Stockton Road, Durham, DH1 3LE
e-mail: e-theses.admin@durham.ac.uk Tel: +44 0191 334 6107
https://etheses.durham.ac.uk

University of Durham

Department of Computer Science

Identifying Reusable Abstract Data Types in Code

Maria Tortorella

The copyright of this thesis rests with the author.
No quotation from it should be published without
his prior written consent and information derived
from it should be acknowledged.

M.Sc.

1994

2 JUN 1995

Abstract

The basic aim of this thesis is to analyse the state of the art in the field of the software reuse

Software reuse is widely regarded as offering the opportunity for improving the software production process. It is expected that a massive increase in software reuse is the most promising way of overcoming the software crisis. It can lead to substantial increases in productivity and also to software systems which are more robust and more reliable.

A Reuse Re-engineering process together with techniques from reverse engineering represent a method to achieve software reuse. A reference paradigm is established to implement the Reuse Re-engineering process. The reference paradigm is composed of five sequential phases, each characterised by the object it produces. This thesis deals mainly with the first phase of the reference paradigm. This phase is called Candidature and it is concerned with the analysis of the source code for the identification of sets of software components that can be candidate to make up a reusable component. Various methods involved in this phase exist in the literature. Each of them has different characteristic and different qualities. One of these approaches is analysed and it is extended in the new method to give a more precise set of reusable abstract data types. In this thesis the new method is presented. A formalisation followed by implementation of it and an evaluation of its quality through quality attributes is given.

Acknowledgments

I would like to thank my supervisor from the University of Durham, Mr. Malcolm Munro, who helped me in the development of the ideas reported in this thesis and supported me throughout their drafting. I am also grateful to him for his intervention in the corrections of this thesis.

I would like to thank Professor Aniello Cimitile from the University of Naples, who helped me to come to Durham, and all the members of the software engineering research group of the *Department of Informatica e Sistemistica* at the University of Naples for the discussions that gave origin to the research idea.

Finally, I would like to thank Professor Keith Bennett and all the members of the *Centre for Software Maintenance* of the University of Durham for all the facilities provided.

Contents

1 Introduction **1**

1.1 Introduction 1

1.2 Criteria for Success 5

1.3 Plan of Thesis 6

2 Existing Reuse Methods **7**

2.1 Introduction 7

2.2 Existing Methods 11

2.2.1 Dominance Call-graph Methods 12

2.2.2 Global Variable Properties Methods 14

2.2.3 User Types as Formal Parameters Properties Methods 18

2.2.4 Logic Methods 21

2.2.5 Program Slicing Methods 22

2.2.6 Testing Methods 26

2.2.7 Similarity Measure Methods and cluster analysis 28

2.2.8 Measurement of Data Binding between Modules Method 31

2.3 Summary 33

3 New Method **35**

3.1 Introduction 35

3.2 The New Method 37

3.3 Formalization of the New Method 42

3.3.1 Abstract Data Types 43

3.3.2 Dominance Tree 44

3.3.3 Formalization 45

3.4 Summary 53

4 Implementation **55**

4.1 Introduction 55

4.2 ADT Method Implementation 56

4.3 New Method Implementation 59

4.4 Summary 63

5 Case Studies **66**

5.1 Introduction 66

5.2 Editor.pas 67

5.3 ExamMarker.pas 75

5.4 Minicalc.pas 81

5.5 Format.pas 87

5.6 Evaluation and Conclusion 91

6 Conclusion **93**

6.1 Introduction 93

6.2 Evaluation of the Criteria for Success 94

6.3 Further Work 96

Chapter 1

Introduction

1.1 Introduction

The last two decades have seen the promotion of software development as an engineering activity, encouraging the use of more rigorously defined software development methods based on sound engineering principles. Such methods have often emerged in response to new ideas about how to cope with the increasing complexity of software systems. The first need to discipline the activities involved in a process of software system development arose in the 1960s with the introduction of third generation computers. These machines were orders of magnitude more powerful than second generation machines and their power made possible the realization of applications that until then were infeasible.

Initial experience in building large sofware systems showed that existing methods of software development were inadequate, not well defined and very often not applicable. There was an urgent need for new tecniques and methods which allowed the complexity inherent in large software systems to be controlled. In those years the term *Software Engineering* was first introduced.

Software Engineering is defined differently by different people. However, the common idea is that Software Engineering is concerned with software systems which are built by teams rather than individual programmers, uses engineering principles in the development of these systems, and is made up of both technical and non-technical aspects [1]. As well as a complete knowledge of computing technique, the sofware engineer should appreciate the problems that the user has in interacting with the software and should understand the project

management problems associated with software production. Moreover, the term 'software' does not simply indicate the computer program associated with some aplications or product, but it includes all the documentation which is necessary to install, use, develop and maintain these programs. In line with this concept, a definitions of Software Engineering suggested by Boehm [2] is: "Software Engineering involves the practical application of scientific knowledge to the design and the construction of computers programs and the associated documentation required to develop, operate and maintain them."

The idea that software development is an engineering discipline led to the view that the process of software development is like the process which has evolved in other engineering process. Thus, a model of software development was derived from other engineering activities and most research into Software Engineering has focused mainly on the development phases of software life-cycle. The initial model was the ***waterfall*** model proposed by Royce [3]. This model underwent various variations till an interative and more complete models, that identify the following five phases [4]:

Requirent analysis and definition is the period in which the requirements, the system's services, constraint and goals are established by consultation with system user;

System and Software Design is the process of representing the functions of the software system, identified in the requirements definitions, in a manner which may readily be transformed to one or more computer program;

Implementation and Unit Testing is the period during which the software design is realized as a set of programs or program unit which are written in some executable programming language. Unit testing involves verifying that each unit meets its specification;

Validation, Integration and System Testing identified the period in which the individual program units or programs are integrated and tested as a complete system to ensure that the software requirements have been met. After testing, the software system is released to the customer;

Operation and Maintenance involves correcting errors which were not discovered in the earlier stages of the life-cycle, improving the implementation of the system units and enhancing the system's service as new requirements are perceived.

In particular, the maintenance phase is one of the most important within the development of

software, also if it has traditionally been seen as the most poor activities compared with the 'more creative' activities of software development, and it has always been assigned to junior programmers as their first task. Next, it has been recognised that maintenance is not just 'bug fixing', but rather a complex amalgamation of activities often having much in common with development. The growth of organization such as the Centre for Software Maintenance of the University of Durham and the increasing number of industrial/academic seminars on the subject indicate that Software Maintenance is becoming a recognized academic and commercial discipline.

Software Maintenance has been defined [5] as: "The modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a changed enviroment".

Software Maintenance activities can be classified into four categories [6]:

Perspective Maintenance enhances the software system by altering and encreasing its functionalities, by responding to user-defined changes;

Adaptive Maintenance is concerned with changements of the software system to adapt it to a change in the data enviroment (system input and output formats), or in the processing enviroment (either hardware of support software);

Corrective Maintenace involves diagnosis and corrections of errors discovered in the system which cause incorrect output or abnormal termination of the software system;

Preventive Maintenance is concerned in the updating the software system to anticipate future problems; this entails improving the quality of the software and documentation, or other software quality factors.

It is estimated that 60% to 70% of the total life-cycle costs are spent on maintenance [7]. In order to make changes, it is necessary first to understand the software and this could involve around 47% to 60% of the maintenace effort. This means that some $30-35\%$ of the total life-cycle costs are consumed in trying to understand software after it has been delivered in order to make changes. A particular branch of Software Engineering that assists the activities of Software Maintenance in the understanding of software systems is the Reverse Engineering. A definition of Reverse Engineering is [8]: "Reverse Engineering is the process to support the analysis and the understanding of data and processing in existing computerized system.

It aims to extract the contents, structure, and flow af data and processes contained within existing software systems in a form that can be enquired and analysed."

Having reverse engineered a software system it may be appropriate to tidy it up, restructuring it to meet current standard, or even re-implementing it in a newer version of programming language, or some other language, and possible on new hardware and operating systems. This complete cycle of Reverse Engineering followed by re-implentation is called Re-engineering.

Reverse Enginering and Re-engineering are two aspects that can help the Software Maintenance.

A way to reduce the time employed from the processes of Software Maintenance can be to produce modularized systems, in which the intervening of maintenance can be localized on single modules instead of on the full system, and to use sofware modules already existing to produce new software systems. In fact, the opportunity to use components already used before and, thus, already tested and maintened should reduce the quantity of time employed in the various activities of the processes of Software Maintenance.

For these reason and for the pursuit of the two objectives of reduction of development costs and the improvement of software reliability Software Reuse is now considered a fundamental aspect of Software Engineering. In fact the adoption of methods for Software Reuse permits both an efficient development of new applications with a raised degree of quality thanks to the availability of checked and tested components, and the re-engineering of existing applications by means of a greater modularity of the software architecture.

In general terms Reuse is defined as "Re-application of source code". Biggerstaff [9] said: "Software Reuse is the re-application of various type of knowledge about a certain system with the aim of reducing the burden of development and maintenance. The reusable elements consist of domain knowledge, development experiences, project choices, architectural structures, specifications, code, documentation and so on".

Reusability is now being taken into account not only during the production of software but in every phases of the development of applications. This incentive to reuse derives from the firm belief that wide use of reusable components is a measure of the industrial maturity of a sector because it shows how much it is possible to keep of an application in successive developments and new realizations. In this way a repository of reusable components constitutes the store that keeps the know-how of the software developer. It can not be said that similar efforts have been carried out in the field of extraction of reusable components from existing systems.

In fact, according to the above, it is a widespread opinion that the reusable components must be planned and not discovered. It is thus reasonable to assume that a significant sub-set of the existing millions of the lines of code must be reusable with little or no modifications.

Then it is obvious to think that the identification of reusable components in existing software systems gives considerable benefits especially when the extracted components are really useful and when the costs for their research and extraction prove less than those for their possible reconstruction. Nevertheless, it is very difficult to quantify those costs since both the real usefulness and validity of the components and the cost for their search are not available in advance.

Many researchers from both the academic and industrial enviroments are interested in Reverse Engineering and Re-engineering activities for the definition and implementation of processes to populate repositories with reusable software components extracted from existing systems. Such processes are called Reuse Re-engineering processes.

1.2 Criteria for Success

The work presented in this thesis will explore one aspect of the Reuse Re-engineering process, that of the identification and the extraction of reusable components. The criteria for success, to be judged in the final chapter, are as follows:

- description and evaluation of existing methods for the identification and extraction of reusable abstract data types;
- development of a more precise method for the identification and extraction of reusable abstract data types;
- formulazation of the new method;
- prototype implementation of the new method to show that it is automable;
- establish a criterion for measuring the effectiveness of the new method.

1.3 Plan of Thesis

The remainder of this thesis is organized as follows.

Chapter 2 deals with some results achieved in the sector of the Software Reuse. In particular, the first section proposes a reference paradigm to set up reuse re-engineering processes. This paradigm, jointly developed within the RE^2 project by the the DIS (Dep. of Informatica e Sistemistica) of the University of Naples and the CSM (Centre for Software Maintenance) of the University of Durham, analyse the relations existing between the Reverse Engineering, Re-engineering and Reuse Re-engineering processes. The second section presents a set of different methods looking for reusable assets in existing code. Each of them utilize a different approach, based on the different kinds of abstractions that are looked for in the existing system. system. The presentation of each method is followed by some discussion to assess the strength and identify the major limitation of the method. The third section proposes a method for the evaluation of the set of modules extracted from an existing system. A set of metrics are proposed for the achievement of that aim.

Chapter 3 proposes a new method looking for extraction of Abstract Data Types in existing code. The first section deals with the starting point of the research and the introduction of a set of quality attributes useful for the evaluation of the quality both of methods for the extraction of reusable modules from code and of the modules. The second section presents the motivations, through the quality attributes, that suggested the extension of an existing method to the new method, and an informal discussion of the new proposition. A rigorously formal presentation of the method is presented in the third section, where all the notions introduced in the previous section are proposed like relations.

The relations introduced in Chapter 3 are trasformed into a Prolog implementation presented in Chapter 4. A presentation of the existing implementation of the starting method is presented. This is followed by the discussion about the new implementation supplying all the information useful for the re-engineering process of the identified reusable components.

Chapter 5 shows the validity of the proposed method through the presentation of four case studies. The first section gives a short introduction to the case studies as applied to method that has been used as the starting point of this work. In the following four sections a description for each case studies follows. Finally the last section shows the evaluation of the quality attributes on the case studies, and discusses the opportunity to use the proposed method as an instrument useful for the comprehension of software systems.

Chapter 2

Existing Reuse Methods

2.1 Introduction

Various paradigms have been proposed to guide all the activities that are concerned with the detection of components that can be reusable, the comprehension and the re-engineering of them according defined standards. A particular paradigm to set up reuse re-engineering processes, that is processes to populate repositories with reusable software components extracted from existing systems, is the RE^2 project [10]. It is an on going research project jointly developed by the DIS (Dep. of Informatica e Sistemistica) of the University of Naples and the CSM (Centre for Software Maintenance) of the University of Durham. The aims of this project are concerned with the exploitation of reverse engineering and re-engineering techniques to facilitate reuse re-engineering processes. In particular it aims to identify the reverse engineering knowledge useful to reuse re-engineering while classifying the theoretical and/or technological open problems for which a basic research effort is required.

Figure 1.1 shows the Reuse Re-engineering reference paradigm developed in the RE^2 project.

In a reuse re-engineering process this paradigm distinguishes the following five sequential phases:

Candidature is concerned with the analysis of the source code for the identification of sets of software components, each of which is a candidate to make up a reusable component. In the next phases, the sets of reusable components identified, to be reusable, have to be de-coupled, re-engineered and generalized. The candidature phase comprehends a

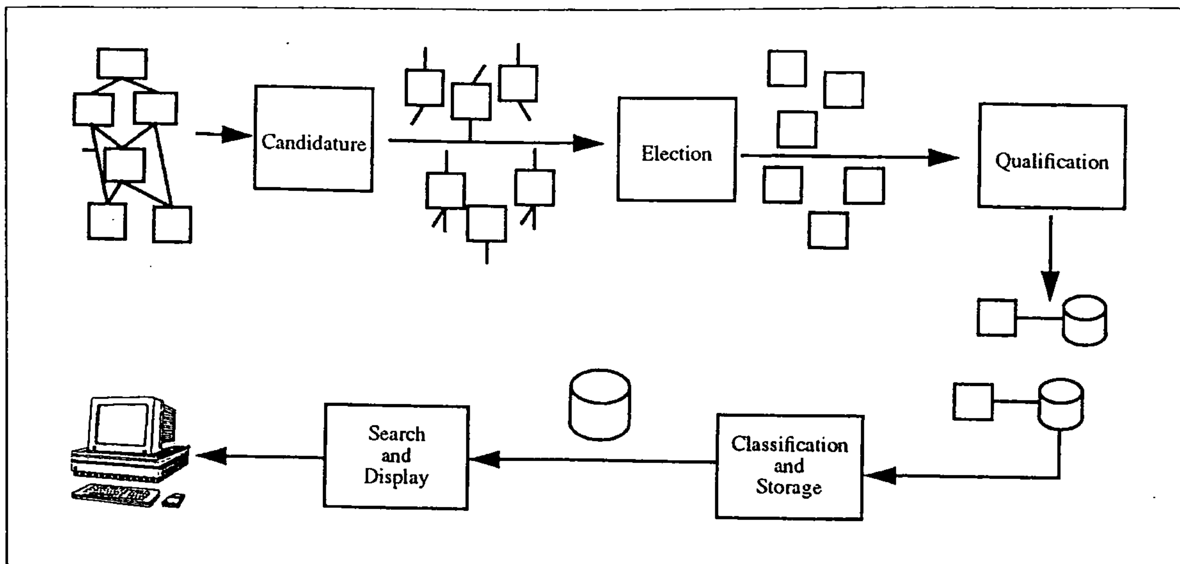

Figure 2.1: The RE^2 reference paradigm

set of activities groupable in three successive steps:

- Candidature Criterion in which a criterion to be applied to produce a first approximation of the set of the reuse candidate modules is defined. This definition, produced by a reverse engineering process, also entails the definition of the model to which to apply the criterion and the information needed to make up an instance of the model. The model instance related to a set of components extracted from code should highlight the kind of these components, that is if they are procedures, functions, slices, data types and so on, and the relations existing among them, that is calls, nestings, declarations, data flow and so on;
- Reverse Engineering in which 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, is defined;
- Criterion Application consists of the application of the candidature criterion to the particular model instance to produce the set of the modules that can be candidate for reuse. Each module consists of software components of the system analysed. In this phase, clearly, they do not yet constitute a reusable module, but they are not yet changed, that is they have not been de-coupled, re-engineered, etc..

Election is concerned with the activities of the analysis of the sets of software components

identified in the previous phase and produces a set of reusable modules. The final set of reusable modules is in reality a subset of the set of modules initially located. In fact, not all the candidate sets will be included in the reusable modules. During that phase some of them will be discarded either because they are too simple to be interesting, or because they are too complex and too expansive in terms of re-engineering operations needed to de-couple and cluster them. The groups of activities that can be identified in this phase are the following:

- Template Definition gives the definition of a module template to re-engineer the reuse-candidate modules. All the software components composing the sets produced in the candidature phase are to be clustered according to this template. The definition of the template have to implement information hiding and interconnection standards. It has to let the module's exportable resources visible to the external world, while encapsulating and making inaccessible the local resources. It has to take into account the characteristic of the source language as regard the primitives for the definition of the modules. It is clear that the definition of the template is simpler in languages that offer syntactic and semantic primitives to define modules, their interfaces and the implementation bodies, like, for example Ada [11] and Modula-2 [12];
- Decoupling entails the definition and the setting-up of a re-engineering process for de-coupling the components from the external environment, i.e. from the components of the subject software system that are not a part of the same module candidate for reuse. The information and the model produced by the reverse engineering step in the candidature phase may help in the definition of the process above, in fact, the setting up of the re-engineering process involves the knowledge of the control and data links existing both among the software components of each candidate module and among them and the components of the old software system. This information is typically produced in the candidature phase to set up the instances of the model;
- Clustering involves the definition and the setting-up of a re-engineering process, but this time, for the clustering of the software components in the template, that is for producing the reusable module.

Qualification is involved in the activities that produce the specifications of each one of the reusable modules obtained in the election phase. Both the functional and the

interface specifications are produced in this phase. Also in this case, the activities can be organized in three steps:

- Specification Model, it is involved in the definition of a specification formalism to express the functionality of a reusable module and how it should be used. That definition is very important in this phase, because the complexity in the following steps depends on it. The effort to produce a descriptive specification and to produce an operational specification is different, and moreover, the effort in producing informal, semi-formal or formal specification is also different;
- Functional Reverse Engineering entails the definition and the setting up of a reverse engineering process to produce the functional and interface specification of the module from the code and to express it according the formalism defined. This process aims firstly to set up low level design document (to understand the functions the module implements) [13] and, secondly, to reach the specification level by means of successive abstractions. The information produced by reverse engineering in the candidature phase, and the document of the old software system, if any, can partially help to reverse engineer to produce the specifications of the reusable modules;
- Testing and Specification Fixing consists of testing and fixing the specifications produced to ensure their correctness and consistency with the code. In this process, even if the test cases of the old system can be helpfully reused, the development of a new testing plan and new test cases is necessary.

Classification and Storage is concerned with the activities that classify the reusable modules and the related specifications according to a reference taxonomy. The aim is to define a repository system and to populate it with the reusable modules produced.

Search and Display is involved in the activities that set up a front end user interface to interact with the repository system. The intention is to make finding the modules the user needs as simple as possible, for example giving them visual support to navigate trough the repository system.

The RE^2 process is mainly concerned with the first three phases in the paradigm, and does not address the last two. These later two phases are related to the setting up of the enviroments to support the reuse of modules rather than to the extraction of these modules from old systems.

2.2 Existing Methods

Various different methods looking for reusable assets in code exist. Each of them has its own peculiarity based on the different kind of abstraction to be looked for. Both the principle on which they are based and the instrument that they use to reach that aim are different for each approach. In spite of this a common feature that can be identified is that each of them produces a set S of candidate modules, and every element s belonging to S may implement a solution of a certain problem. If this problem presents itself with a high frequency in the application domain the module must be reengineered and qualified, transforming it into a reusable module and enclosing it into the set R of reusable modules.

Starting from the set S, it is possible to obtain the set R by following two steps:

- Producing a sub-set $S' \subseteq S$ containing only the candidates elected for reuse

$$S' = sel(S)$$

- Re-engineering and defining a qualification process p to produce the following set R

$$R = \{r \mid r = p(s) \ \ \forall s \in S'\}$$

The function $sel(S)$ is fundamentally a *Concept Assignment Process.* It requires the intervention of a software engineer with the peculiarity that the major the quality of the adopted method, the less the effort of the software engineer to assign a concept to the candidate modules.

The quality of a candidature criterion is measurable by means of quality attributes. In the next chapter some quality attributes will be introduced and they will be used to evaluate the quality of the approach proposed in this thesis.

In this chapter various examples of candidature criteria are presented. The aim is to provide a panorama, as exhaustive as possible, of structural methods proposed for the process of Reuse Re-engineering. Each subsection will examine a different approach. The proposed methods are based on the following:

- dominance call-graph;

- global variables;
- user types of formal parameters in procedures;
- structural model of a software system based on logic;
- program slicing;
- testing;
- similarity measure methods and cluster analysis;
- measurement of data binding between modules.

In the process of software reuse the phase of Candidature comprehends a phase of evaluation of the identified reusable components. In the next chapter some quality attributes to the evaluation both of the methods and the components identified will be introduced.

2.2.1 Dominance Call-graph Methods

Cimitile and Visaggio [14] propose a candidature criterion, that searches for functional abstraction in existing system. The method uses the call-graph and its transformation into the dominance tree and then it carries out an analysis of these structures. This criterion is articulated in four different candidature rules.

Before setting out these rules it is necessary to give some definitions:

The *call-directed-graph* of a program is defined as a graph $CDG = (N, E)$ where $N = \{s\} \cup PP$ is the set of the main program s and of the set PP of all procedures and functions, and E is the Call Relation $(\{s\} \cup PP) \times PP$.

The *call-directed-acyclic-graph*, denoted as CDAG, is obtained from CDG by collapsing every strongly connected subgraph into one node.

According to Hetch [16] 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 spans 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 reflexive and transitive dominance relation of CDAG is a tree, called *Direct Dominance Tree.*

The *Strong Direct Dominance Tree,* denoted by SDDT, can be obtained from the Direct Dominance Tree by marking all the edges that connect two nodes that are in a relation of strong direct dominance.

The set S of all subtrees of a SDDT is composed of the set MET of all subtrees having only marked edges, and the set UMET of all subtrees having both marked and unmarked edges.

The *Reduction of the Strong Direct Dominance Tree* is a tree and it is denoted by RSDDT. It can be obtained from the SDDT by collapsing all the subtrees MET to a unique node.

Given these definitions the four rules can be enunciated:

Rule 1: by examining the CDG it can be determined if there are strongly connected subgraphs. The program units associated with the nodes of each subgraph are extracted to constitute a candidate module for reuse.

Rule 2: by examining the SDDT it can be determined if there is some subtree MET. In this case the program units associated with the nodes of each subtree are extracted to constitute a candidate module for reuse.

Rule 3: by examining the SDDT it can be determined if there is some subtree UMET. In this case the program units associated with the nodes linked to the subtree root by a marked edge can be extracted to constitute a candidate module for reuse in a relation of "USES" with the program units associated with the nodes linked to the root by an unmarked edge.

Rule 4: by examining the RSDDT a marked edge constitutes a "COMPOSED OF" relation between the modules associated to the nodes that the edge links, while an unmarked edge represents a "USES" relation.

A method to search for functional abstractions in a software system and based on the dominance tree can be applied to software systems coded in procedural languages and designed according to the principles of the modular programming and functional decomposition.

Therefore such a criterion applied to software with different characteristics (for example with object-oriented structure) would not succeed in proposing significant reusable modules. Experiments using the dominance criterion, performed on a software system that satisfies the above requirements confirmed the relevance of this method [15]. The criterion was verified with a sample of approximately 15,000 lines of Pascal code extracted from a total of 300,000 lines of an existing system. The sample consisted of 25 programs with 424 procedures and functions. From this set of program units, 169 candidate modules were identified. The results showed that the criterion aggregates program units involved in the implementation of functional abstractions with an Adequacy of almost 35%. The real reusability of the software was almost 50%. A large number of the modules (almost 43.8%) were discarded because they were composed of only one procedure or function. It was seen that these program units were not aggregated because they were procedures and functions involved in abstractions different from functional abstraction (such as, for example, data abstraction). This result shows the importance of the choice of a candidature criterion in relation to the characteristics of the analysed software system. If these modules were not taken into account, then the percentage of successes increases considerably.

It is possible to complete the criterion with an analysis on the data coupling between procedures and functions and to use the results of this analysis to propose a more complete measure of the reengineering effort to isolate the selected components and encapsulate them in a module for reuse.

2.2.2 Global Variable Properties Methods

In [17], a criterion that searches for candidate objects for reuse is presented by Liu and Wilde. The authors say that an object is characterized by a tuple $O = (F, T, D)$ where F is the set of all program units, T the set of the data types and D is set of data items. Any of these sets can be empty.

The proposed method provides candidate objects composed of global variables and program units that use them. In particular the approach is articulated in three steps:

Step 1: for each global variable x, $P(x)$ indicates the set of program units which use it;

Step 2: supposing that each $P(x)$ corresponds with a node in a graph, a graph $G = (V, E)$ can be constructed such that:

$$V = \{P(x) \mid x \text{ is shared by at least two routines}\}$$
$$E = \{(P(x_1), P(x_2)) \mid P(x_1) \cap P(x_2) \neq \emptyset\}$$

Step 3: looking at the graph G, it can be verified if strongly connected subgraphs exist, and in this case the program units associated with the nodes of each subgraph can be extracted to form a candidate object for reuse and composed of those units and relative global variables. If $C = (V_c, E_c)$ indicates a strongly connected component the object extracted from it can be represented as a tuple (F, T, D), where:

$$F = \bigcup_{P(x) \in V_c} P(x)$$
$$T = \emptyset$$
$$D = \bigcup_{P(x) \in V_c} \{x\}$$

This criterion was experimented with by using some programs written with conventional programming languages such as C, Ada, Cobol and Fortran and significant results were obtained. In this case, it was necessary for the software engineer's intervention to resolve conflicts and provide knowledge about the application domain. This point will be discussed below.

In [18], Dunn and Knight propose an expert system to locate potential reusable modules. The knowledge base of the system is based on four forms of knowledge: knowledge about the application domain and design of the software that must be examined, knowledge about the domain in which the extracted modules have to be reused, metric definitions and reengineering knowledge. The system is based essentially on three functional elements:

- a C parser, that generates the abstract syntax tree from C source code;
- a Prolog interpreter, that identifies the candidate components for reuse;
- an interactive interface, that allows the communication with the system.

The Prolog interpreter searches for reusable components in three different ways:

- by identifying, by the analysis of the call-graph, the program units that are invoked more than once;

- by identifying the program units that are strongly connected by analysing the various types of couplings. The types of coupling used by Dunn and Knight are data coupling, when the program units share simple data as formal parameter; common coupling, when the program units share global data; external coupling, when the program units share data with the external world; control coupling, when the program units share data used for control. The program units that are loosely bound are easier to remove and use in other contexts than those that depend heavily on other functions or non local data. After this the various kinds of couplings between the components of the system are recognized and the program units that are not connected can be candidate for reuse. The study of this last point is not completed and the work is continuing to determinate how the restriction of the above coupling characteristics can be relaxed so that sets of program units with varying coupling degrees can be identified and be candidate for possible reuse.

- by identifying program units and global variables that can be grouped together to form abstract data types. The program is represented as a bipartite direct graph in which the nodes are the names of program units or the names of global variables, and the edges are directed from nodes representing program units to nodes representing global variables specifying a "USES" relationship. The criterion operates by using a depth-first visit of the graph to find the strongly connected subgraphs. Each of these subgraphs can represent a candidate module for reuse.

This expert system was tested with 5 public-domain software systems, written in the C language. Some evaluations were done on the base of criteria like *practicality* (how useful a part would be in an application either in the same problem domain or in other domains), *reusability* (how much effort is necessary to reengineer a part in order for it to be reasonable to propose it for reuse) and *understandability* (how difficult it is to comprehend what a reusable candidate part does). In all cases the obtained results were satisfactory.

The two algorithms described above produce fundamentally the same results. These results are satisfactory when the system to analyse has a fully object-oriented structure for which it is possible to identify all the objects constituting the system. The situation is different when object-like components are mixed with functional-oriented components.

Livadas and Roy propose in [19] another criterion based on the properties of the global

variables that brought a greater level of granularity. The method in question is known as *rboi* (receiver based object identification) and is founded on the operation of definition of a same global variables from a number of program units, where a program unit defines a global variable if there exists at least one execution path in it where the variable is modified. The *receiver* is defined as triple $(F, T, rboi)$ where F is the set of the program units that *define* variables of type T. In reality, in this case, also the pass-by-reference parameters are indicated with the words 'global parameter'.

Essentially the criterion consists of the clustering of the types of the global variables with all the program units that define them to form some candidate components for reuse. It is clear that the program units that only use the variables without modifying them are not present in a cluster that is constituted as above.

This method has been tested with small programs and showed good results, but the problem remains for medium and large programs. In fact, no help is provided to distinguish the program units that implement methods of the same object from the program units that access data items belonging to different objects, and so the algorithms produce results of low quality and different objects are clustered in a unique candidate module. The problem is to identify these undesired links and remove them.

Two types of undesired links exist: coincidental connections and spurious connections. Coincidental connections are due to program units that implements more than one function, each belonging to a different object. This kind of connections can be removed by slicing each interested program unit and obtaining one program unit for each elaborated function. Spurious connections are due to program units that implement one specific operation by accessing to the structure data of more than one object. Such kinds of connections can be removed by deleting the program unit.

A suggestion to solve the problem of the undesired links is proposed by Canfora, Cimitile, Munro and Taylor in [24]. Their approach is based on a graph called *variable-reference* graph, that is essentially the same graph adopted by Dunn and Knight, and on the definition of internally connected subgraph.

A internally connected subgraph of a graph G is a graph $g \sqsubset G$ such that the number of edges that connect couples of nodes belonging to g is higher than the number of edges linking nodes in $G - g$. The internal connectivity of a sub-graph g is measured by an index IC_g:

$$IC_g = \frac{\#of\ edges\ that\ link\ couples\ of\ nodes\ in\ g}{\#of\ the\ edges\ that\ have\ at\ least\ one\ vertex\ in\ g}$$

If the number of the individual links is sufficiently low, internally connected subgraphs can be identified in the graph and each of these subgraphs can correspond to a candidate object.

The proposed algorithm tries to identify reusable objects through an iterative process. The first step is to associate each program unit f with an index IC_f that measures the internal connectivity of the subgraph generated by clustering together all the data items referenced and all the program units that access only these data items. IC_f is the ratio between the cardinalities of the set of edges that have either vertexes in the subgraph, and that of the set of edges with at least one vertex in the subgraph. It is evident that if f generates a strongly connected subgraph the value of IC_f is equal to one.

Using the value of IC_f the value of ΔIC_f can be calculated as the difference between the internal connectivity of the subgraph generated by f and the sum of the internal connectivity of the subgraphs generated by the other program units in the above subgraph. The program units with a sufficiently high value of ΔIC_f can be used to generate clusters. If some program unit exists that links together two subgraphs with high internal connectivity, it corresponds to a coincidental or spurious connection and so it can be sliced or deleted.

With the new situation the process restarts and continues until a graph with all isolated subgraphs is obtained.

2.2.3 User Types as Formal Parameters Properties Methods

In the sphere of the methods based on data abstraction the second criterion proposed by Liu and Wilde in [17] is included. The reusable candidate objects produced are composed of program units and user defined data types.

The five steps which articulate this criterion are the following:

Step 1: a topological order of all types in the program is defined so that the user defined data type t_1 is a *sub-type* of t_2 if t_1 is used to define t_2, in the same way t_2 is a *super-type* of t_1;

Step 2: a relationship matrix $R(F, T)$ is constructed. In this matrix the rows are associated with the program units and the columns are associated with the user defined data types. The generic elements $R(f, t)$ is set to 1 if the type t is a sub-type of one of the formal parameter or of a return value of the program unit f. All the other elements are set to 0;

Step 3: for each row f the element $R(f, t_1)$, set to 1 in the first step, is set to 0 if an element $R(f, t_2)$ marked as 1 and with t_2 super-type of t_1 exists;

Step 4: all the program units sharing the same type are collected in a same group. Specifically the two program units f_1 and f_2 are grouped together if a type t exists so that $R(f_1, t) = R(f_2, t) = 1$;

Step 5: a reusable candidate object with the program units and the involved user types is constructed from each group. Exactly one candidate object $O = (F, T, D)$ is formed from each group where:

$$
\begin{aligned}
&F = \{f \mid \text{the program units } f \text{ is a member of the group }\}\\
&T = \{t \mid R(f, t) = 1 \text{ for some } f \text{ in } F\}\\
&D = \emptyset
\end{aligned}
$$

This criterion was experimented with for programs written in Ada, Cobol, C and Fortran languages. The results often showed a candidate object that was too big for the grouping caused by some type improperly used in several program units. Again, in this case the intervention of a software engineer proves useful to isolate that type and reorganize the object in more objects of smaller size.

Another method operating with analogous presupposition is presented in [10] by Canfora, Cimitile and Munro. This method consists of a set of direct relations obtained from the static analysis of the code and it involves the construction of a directed graph that shows the relation existing between the program units and the user defined data types that are used to declare the formal parameters. In the construction of this graph the definition of topological order introduced in [17] is used.

The graph $G = (N, E)$ is such that N is the set of the nodes associated either with a program unit or a user defined data type, and E is the set of the couples (c, t) such that c is a node associated with a program unit and t is a node associated with a user defined data type

and used from c to declare a formal parameter. In the next step this graph is simplified by eliminating all the edges (c, t) for which a type t_1 that is super-type of t and is such that the edge (c, t_1) of the graph exists. Finally in the obtained graph each subgraph represents a candidate component for reuse.

The experimentation on software systems written in Pascal showed results that can be compared with those obtained in the above method by Liu and Wilde. In both the cases the large use of global variables, limiting the use of formal parameters to change information between the program units, induced the creation of modules that were too simple or not very significant. It can be useful to think of methods that consider both of these aspects, the global variables and the formal parameters. In other cases it can verify that the large use of some user defined data types causes the grouping of more modules, and the intervention of a software engineer becomes very important.

This method will be analysed more deeply in the next chapter and the results, obtained by the application of it to some case studies, will be discussed, and a solution to obtain better results will be proposed.

Haughton and Lano in [20] illustrate a process for the normalization of objects obtainable from existing code written with an imperative language. In particular this approach can be included in the other section on the object orientation and in the identification of abstract data types.

The method uses the notation Z++ [21], and it can be synthesized as follows:

Step 1: to identify all abstract data types and all their corresponding operations.

This process is not completely automatic since it is necessary to have the knowledge of the application domain. However it can be eased by making use of heuristics in the identification of data types. This involves the determination of the data structures that are present in an application and that are not primitive but are composed of primitive entities. This determination leads to the identification of the operations that work on those data structures;

Step 2: to group all segments of code that have been identified because they perform some operations on a specific data type.

This step is useful to make clear the data type and all the operations that take effect on it and perform a partial restructuring of the original code in various segments of the code. This restructuring is only partial because the code is only changed and permuted taking no account of the syntactical and semantic bond.

Step 3: to describe the objects that are present in the code using the Z++-notation;

Step 4: the objects identified in the previous steps are normalized.

2.2.4 Logic Methods

In [22] Canfora, Cimitile and Munro present an approach based on logic for the definition of candidature criteria for the search of abstract data types.

This method is synthesized as follows

If $STYP$ is the set of the couples (c, t) such that c represents a program unit and t represents an user defined data type used to define a formal parameter and such that c does not use a super-type of t, the following relations are defined:

$$ABTYP = (trans(STYP)STYP)^*$$
$$CCTYP = (trans(STYP)STYP)^* trans(STYP)$$

where $trans(R)$ and R^* indicate the transpose and the reflexive transitive closure of the relation R.

The relation ABTYP defines the supporting structure of the abstract data type, that is the user defined data types that contribute to the the constitution of the candidate module for reuse. The relation $CCTYP$ defines the operators of the candidate abstract data type, that is the program units that have to be included in the candidate module.

For the logic nature of the criterion a logic programming language like Prolog has been used to implement a prototype tool. This tool showed itself to be particularly flexible and easy to develop, and these are very important characteristics for the evolving nature of a prototype tool operating in the field of reuse re-engineering whose knowledge and technologies are not stable but continuously changing.

The tool, operating on systems written in Pascal, achieved satisfactory results [23] that can be compared with those obtained with the abstract data type methods. Also in this case human knowledge and heuristics were necessary to identify the coincidental and spurious connections possibly existing among the components of a candidate abstract data type, thus improving its quality.

2.2.5 Program Slicing Methods

Ning, Engberts and Kozaczynsky propose in [25] various methods based on the technique of program slicing supporting the recovery of reusable components from existing code. The segmentation of the program is reached in two steps: *focusing* and *factoring*.

The operations of focusing have the aim to identify the portions of reusable code. In particular, there are five methods to select the segments:

- **Select Statements**, with which, without particular bonds, the user can include statements in a segment in an arbitrary way. A powerful syntax-directed browser guarantees the syntax validity of the choices, giving the possibility to select only complete statement objects;
- **PERFORMed Statement**, based on the consideration that a call hierarchy reflects a functional decomposition of a program. If the user chooses a 'procedure call' statement, all the code relative to the called program unit is automatically included in the segment. If there are any 'procedure call' statements in the new portion of code the process starts again and continues till the complete 'explosion' of all this kind of statements;
- **Condition-base Slice**, founded on the consideration that many functions in a business field are structured along conditional tests and can be potentially useful to identify areas of a program reachable under a globally specified condition. The user specifies a logical expression and a 'slicing range' (in terms of where to start and end slicing in the program) and automatically all the reachable statements along a control flow path for which the given expression is true are included in a segment.;
- **Forward Slice**, based on the search of areas of code depending on values of the variables given in input to the analysing program. Given a variable and a 'slicing range' all the statements that can be potentially affected by this variable are inserted in a segment. A variable affects a statement or as *data flow* if the statement uses the

value of the variable, or as *control-dependence* if the execution of the statement depends on the value of the variable (in this case the variable is a part of a logic expression in a conditional statement). The process is recursive because when a statement is included in a segment all the variables that it contains are in turn used as slicing variables.

- **Backward Slice**, comes from the observation that it is interesting to look for areas of code that contribute to the production of the output of the analysed program. Given a variable and a 'slicing range' all the statements that can potentially affect the value of the variable are inserted in a segment. In the same way as the forward slice, a statement can affect a variable either in terms of data-flow or in terms of control-dependence. In this case too it is possible to proceed in a recursive way by using the new variables that are present in the new statements.

The factoring step is used to extract the segment of code and cluster them in reusable modules. Besides the statements identified are provided of a set of information that are essential for the completeness and the independence of the module. This information includes the variables referenced in the sections of code selected with the respective *Data Divisions* that are extracted and attached to the module, a new *Identification Division*, a new possible *Linkage Division* if the module created will be a subprogram, and other information.

The methods synthesized above led to the realization of a tool useful to extract reusable components from large legacy software systems written in Cobol. In particular this tool has been used for the transformation of these systems from an obsolete hardware platform to new client-server architectures. The authors have not, however, given the results on the real reusability of the code so extracted.

The slicing techniques allow the extraction of functionalities implemented in the code. The slicing is useful especially with code written using languages that do not have explicit procedural syntactic mechanisms by means of which it is possible to encapsulate functionalities (the Cobol language is one of these languages).

One of the focusing techniques proposed by Ning, Engberts and Kozaczynsky comes from the supposition that a 'perform' statement can be logically made equal to a procedure call statement of a structured programming language, like C or Pascal. Consequently every block of instructions composing the body of a 'perform' is proposed as a reusable module that implements a functional abstraction. So the supposition of this technique is that a hierarchy

of calls understands a functional decomposition of the software system, therefore the software components (statements and relative data) involved in a hierarchy of call relations can be joined in reusable modules.

A weak point of the criterion shows itself when a 'perform' is activated in several points of the program. In this case the statements and relative data corresponding to this perform will be joined in different modules, with the consequence that parts of code candidate for reuse are in different modules.

Another slicing approach is that of Cutillo, Fiore and Visaggio [26]. In particular, the study presented addresses the problem of the identification and the extraction of "domain independent" components from large programs having an incomplete project documentation. The search is based on the differentiation between two fundamental classes of components, those depending on the program domain and those depending on the implementation technological platform. The distinction between these two kinds of components is useful for various reasons such as the major maintenance caused by the isolation of the code subject to the technological changes; the major readability of the code since the parts that relate to the application domain and to the interface are more evident and the increased possibility of software migration between the different platforms.

The components that are independent of the application domain, called also structural components, often depend on typical decisions taken in the design phase. These decisions often disappear from the programs and existing documentation following modifications of the system during its life cycle caused by poor updating of the documentation. The structural components are often scattered inside the program among the code lines of functional components and normally the modifications to one class have reasons and approaches different, and in the same time not independent, from those of the other class. So it is very important for the identification of the structural components, and the authors emphasize the importance of the techniques of slicing for the extraction of reusable components. They assert that the application of the slicing has to be led from the data of the system and propose an approach founded on *Direct Slicing*, a software decomposition technique broadly documented in [27]. This slicing technique, by considering the results of a preliminary phase of data recovery, is applied to an opportune subset of the input and/or output data of the program.

The method to extract the above mentioned components can be so synthesized:

step 1: extraction of a direct slice with a suitable criterion for each instruction handling a

structure (File, etc.);

step 2: analysis of each slice obtained to establish its validity as a reusable module;

step 3: refinement of each slice if the resulting module is too complex;

step 4: constitution of a module combining the slices obtained in the previous steps;

step 5: determination of the complement of the extracted slices with respect to the program;

step 6: composition of the structure chart with the modules obtained in step 4 and the MAIN resulting from the step 5;

step 7: repetition of the previous steps on MAIN, after having identified another extraction point and corresponding criterion.

The parts of the program that can be isolated after step 5 remain in the MAIN program until they are eliminated, or until they are not considerated in the subsequent iterations. However, afterwards, the candidate modules have to be analysed by a software engineer who will have to purge from the code those parts that are useless for reuse.

The objection that can be made of this method is that it is not always possible to start a data recovery process on a software system before the application of the slicing, in fact in the large part of the cases the slicing technique is used just as tool to identify the functionalities implemented in a system.

The method was tested with various Cobol programs with a monolithic structure that used broadly conditioned and unconditioned jumps. Although the obtained results were positive, it is opportune to observe that this method can lead to slices that are too complex to be easily understood or too simple. In these cases it is possible to iterate the slicing process until the various sub-functionalities aggregated in the initial slice are identified, or, on the contrary when the slice extracted is too simple to encapsulate the slice obtained in another slice that uses it. In any case the intervention of the software engineers is important to solve the ambiguity.

Points for future development of this method can be identified. In particular the extension of the experiment to other kinds of software, a major definition of the reverse engineers' task in sight of a full automation of the process and a major effort for the real reuse of the extract components.

2.2.6 Testing Methods

In [28] Wilde and Gomez describe a method founded on the extraction of user functionalities from existing code. In particular this method intends to identify the code implementing a functionality through the use of test cases.

The problem of functionality identification can be seen, fundamentally, as the identification of the relation existing between the way in which the program is seen by the user and the way in which it is seen by the programmer. The user sees the program as a set of functionalities:

$$FUNCS = \{f_1, f_2, ..., f_n\}$$

wheras the programmer sees it as a set of software components:

$$COMP = \{c_1, c_2, ..., c_m\}$$

So the problem is the identification of a relation $IMPL$ on $COMP$x$FUNCS$ that enumerates the sets of components that implement a certain functionality. This relation can be detected by means of the application of test cases: a test case T_i exhibits a set of functionalities $F(T_i) = \{f_{i,1}, f_{i,2}, ...\}$ and, at the same time, it effects the testing on a certain set of components $C(T_i) = \{c_{i,1}, c_{1,2}, ...\}$.

By starting from these suppositions two different approaches to obtain the correspondence between the components and the functionalities can be formulated: *probabilistic* and *deterministic*. The former, by starting from a sample of test cases extracted from a population composed of every possible test sequences for the program, tries to identify the *best indicators* of a given functionality. If $p_{f,c}$ indicates the conditional probability that a test case used to test the component c displays also the functionality f, it can be written:

$$p_{f,c} = P(f \mid c) = P(f, c)/P(c)$$

Clearly the *best indicators* components of a certain functionality are those for which the value of $p_{f,c}$ approaches 1, since if c is performed then a major probability that f is present exists.

Supposed that $T_1, T_2, ...$, are a random sample of test cases, an approximation of the $p_{f,c}$ can be obtained from:

$$\hat{p}_{f,c} = Freq(f,c)/Freq(c)$$

Finally the relation $IMPL$ can be constructed as:

$$IMPL_z = \{c : COMPS; f : FUNCS \mid \hat{p}_{f,c} \geq z\}$$

where z is a threshold value with $0 \leq z \leq 1$.

In the second approach the relation $IMPL$ can be constructed, for a certain functionality f, by collecting all those components that exhibit it when they are tested and rejecting all the other components that do not exhibit it. In practice for some test case T can be written:

$$IMPL'(c,f) \Longrightarrow c \in C(T) \text{ and } f \in F(T)$$

and there would be no T' such that

$$IMPL'(c,f) \Longrightarrow c \in C(T) \text{ and } NOT(f \in F(T'))$$

The relation $IMPL'$ is a limiting case of $IMPL_z$, in fact it can be demonstrated that:

$$IMPL_{1.0}(c.f) \Longleftrightarrow IMPL'(c,f)$$

Therefore the deterministic approach is weaker than the probabilistic one.

The method was experimented by using a program written in the C language and containing about 15,000 code lines in 360 functions. The program was several years old and had been modified a number of times, and so it can represent the typical program that a real world maintainer might approach. The first difficulty was to define the kind of components. Each program unit can be associated with a component, but in this case a program unit is a too gross a level of detail to locate a specified functionality from test execution data. A different definition can be a program statement, but here a functionality can be implemented from an execution path and not from a single program statement. After these considerations it

was decided to use each decision point, that is each result from an *if* or *while* predicate and each possible result from a *switch* statement, as a component.

The Probabilistic approach achieved slightly better results than the Deterministic one, identifying at least the same components of the latter together with other components.

2.2.7 Similarity Measure Methods and cluster analysis

In [29] Schwanke illustrates a model to obtain the modularization of the existing code for improving its quality.

In particular, they define a heuristic ***design similarity measure***, based on Parnas' information hiding principle[1] [30]. Two approaches are identified. One approach, the *clustering*, permits the identification of sets of procedures that share a sufficient number of design information in order to place all the procedures of each set in the same module. The other approach, the *maverick analysis*, is useful for the identification of single procedures that have been selected in a wrong module, sharing more information with procedures present in other modules than with those included in their own module.

Supposed that A,B,C,... are objects described by sets of features a,b,c,... respectively, the *design similarity measure* defined is based on the following:

$a \cup b$	is the set of features that are common to A and B;
$a-b, b-a$	is the set of features that are in A and not in B and *vice versa*;
w_{a_i}	is the weight of the single feature a_i of a;
$W(a) = \sum_{a_i \in a} w(a_i)$	is the weight of the set of features a;

$$Linked(A,B) = \begin{cases} 1 & if\ A\ calls\ B\ or\ B\ calls\ A \\ 0 & otherwise. \end{cases}$$

[1] In 1971, Parnas wrote of the information distribution aspects of software design "The connection between modules are the assumption which the modules make about each other". Next, he formulated the ***information hiding criterion*** advocating that "a module should be characterized by a design decision which is hidden to all others, and its interface or definition should be chosen to reveal as little as possible about its inner working".

The similarity function is defined as follows:

$$Sim(A,B) = \frac{W(a \cup b) + k * Linked(A,B)}{n + W(a \cup b) + d * (W(a-b) + W(b-a))}$$

In this function it can be noted that:

- all coefficients are non-negative;
- only the shared and distinctive features are important. The constant d checks the relative importance of the shared and distinctive features;
- similarity increases with the shared features and decreases with those distinctive;
- similarity is zero if there are no shared features and no procedure calls to other modules;
- the constant n checks the normalization;
- $Sim(A,B) = Sim(B,A)$.

The values to assign to k, n and d are selected on the basis of empirical choices by trial and error. In a future evolution of the methods, these constants will be defined in an automatic way.

On the basis of this definition, the clustering algorithm, called *hierarchical, agglomerative clustering* proceeds as follows:

Place each procedure in a group by itself
Repeat

Identify the two most similar groups
Combine them

until

the existing groups are satisfactory

The groups obtained in the end of the process are used to define the components of the module.

The algorithm presents three variations:

batch clustering does not take into account the intervention of the software engineer. Binary trees representing the results of *combine* operations are created, and from them, in a heuristic way, the branches considered useless are eliminated. With this variation, the results obtained are often very different from those the software engineer would like;

interactive, radical clustering considers that the results of a *combine* operation are submitted to the software engineer. The acceptance or the rejection of a combination influences the choices that are adopted in the next step in the algorithm;

interactive reclustering uses a previous classification to guide the clustering. It starts by noting the original module in which each procedure is located. Successively, as two groups are selected to be combined it is checked to see if their members were all in the same module. In this case, they are combined without requesting the confirmation of the software engineer; otherwise, the engineer can accept or reject the combination, or can set aside one or both the groups for a future classification.

In the clustering process, the procedure are classified as:

- *Subject*: a procedure that is being compared with a number of other procedures for clustering or classification;
- *Neighbour*: a neighbour of a subject is a procedure with which it has at least one common feature;
- *Good Neighbour*: the good neighbours of a subject are those neighbours that belong to the same module of the subject;
- *Bad Neighbour*: the bad neighbours of a subject are those neighbours that do not belong to the same module of the subject.

The approach of *maverick analysis* is useful to identify potential *mavericks*, that is procedures located incorrectly. The potential *mavericks* are identified by searching the most similar neighbours of each procedure and noting the modules they belong to. In a formal way, it is possible to say that a *maverick* is a procedure for which the large part of its k nearest neighbours are *bad neighbours*.

Finally, both the approaches present the software engineer with a list of suggestions useful for the modularization of the code.

The method has been realized in a tool called **Arch.** It has been used in experiments on 5 different software systems and it has been demonstrated that it can provide good support for the software engineer in real cases.

2.2.8 Measurement of Data Binding between Modules Method

The method proposed by Hutchens, Delis and Balisi [31, 32] is based on the measurement of the data and type bindings between modules and components. The data bindings provide a measure of the interaction existing between either components or modules. In particular, they provide the "proximity" of a system components, and this measure of "closeness" is input to a method of analysis of mathematical taxonomy suitable for the construction of a simple tree diagram of the involved elements. This diagram, called a *dendrogram*, expresses the similarities and the dissimilarities between the elements.

The data binding methods are an example in the category of visibility data methods. The definition of the data bindings is the following:

> Let α and β two program segments and γ a variable global to α and β. If γ is defined from the segment α and used from the segment β, then there exists a data binding between these two program segments, and it is indicated from the triplet (α, γ, β).

The triplet describes the flow of information between the first segment and the second. It is possible that the reverse binding exists if a global variable γ' exists from which the triplet (β, γ', α) exists.

Several families of data binding exist. The previous definition is referred to the ***actual data binding***. Other kinds of data binding are: the ***potential data binding*** that expresses the possibility that two segments communicate through a variable that is located in their lexical scope; the ***use data binding*** that indicates that two segments use, either in reference or assignment, a variable included in their scope; the ***control data binding*** that expresses a requirement of an extra condition on the basis of which the control can pass from the segment α over the segment β.

By exploiting the data binding concept, it is possible to use a mathematical taxonomy to group similar objects. The similarity existing between two objects is based on the properties

of the objects. Supposed to have n objects, a dissimilarity matrix for them may be computed from the binding matrix. This is a matrix with dimension nxn as follows:

$$M = \begin{pmatrix} \mu_{1,1} & \mu_{1,2} & \cdots & \mu_{1,n} \\ \mu_{2,1} & \mu_{2,2} & \cdots & \mu_{2,n} \\ \cdots & \cdots & \cdots & \cdots \\ \mu_{i,1} & \cdots & \mu_{i,j} & \mu_{i,n} \\ \cdots & \cdots & \cdots & \cdots \\ \mu_{n,1} & \mu_{n,2} & \cdots & \mu_{i,n} \end{pmatrix}$$

Element $\mu_{i,j}$ indicates the number of control flow data bindings existing between the objects i and j of the form (i,x,j) or (j,x,i) for some variable x.

By starting from matrix M the dissimilarities matrix N of dimension nxn, also called 'distance matrix', can be constructed. This matrix is passed to an iterative algorithm that determinates the 'objects' dendrogram'.

The computation of the matrix N exploits the notion that if a component of the system is entirely connected to just one other component, that connection should be computed as a lower dissimilarity than any connection that is not complete. More specifically, the matrix N expresses the percentage of the data bindings that connect to either of the two components and are shared by the two components. The matrix M is symmetric and the element $N_{i,j}$ of N is determinated from the following formula:

$$N_{i,j} = \frac{\sum_k \mu_{i,k} + \sum_k \mu_{k,j} - 2\mu_{i,j}}{\sum_k \mu_{i,k} + \sum_k \mu_{k,j} - \mu_{i,j}}$$

.

The value $N_{i,j}$ indicates the distance between the object i and the object j. Since:

$$\sum_k \mu_{i,k} + \sum_k \mu_{k,j} - \mu_{i,j}$$

is the number of data bindings in which either i or j occur and

$$\sum_k \mu_{i,k} + \sum_k \mu_{k,j} - 2\mu_{i,jthedi}$$

is the number of data bindings in which either i or j occur but not both, $N_{i,j}$ is the probability that a random data binding chosen from the union of all bindings associated with i or j is not in intersection of all bindings associated with i and j. If the objects i and j have no external connections then

$$\sum_k \mu_{i,k} = \sum_k \mu_{k,j} = \mu_{i,j}$$

and $N_{i,j} = 0$. Moreover, if i and j share no common data then $\mu_{i,j} = 0$ and $N_{i,j} = 1$.

At this point, the clusterization process proceeds in a bottom-up fashion. In each iteration, a series of successive fusions of the n objects into clusters is made to progressively reduce the dimension of the distance matrix. The objects that are first grouped are those with a smaller distance, and, through the clusterization, they form a new object that includes all of them. In the next iteration of the process, the set of objects to consider is the previous set with the substitution of the clusterized objects with their clusters. The distance between a cluster object and an object is calculated by choosing the minimum distance between each object in the cluster and the object. The iterative process terminates when it is not possible to clusterize other objects.

Based on the concept of the data binding, a tool for Ada source reusability has been constructed. One recognised limit of the method is that the cluster analysis is based on a limited knowledge of the object and not based on very sound probability models.

2.3 Summary

In this chapter the RE^2 reference paradigm has been presented. It has been developed within the RE^2 project, an on going project jointly developed by the DIS (Dep. of Informatica e Sistemistica) of the University of Naples and the CSM (Centre for Software Maintenance) of the University of Durham.

The reference paradigm proposes a systematic approach to analyse an existing software system for the identification of reusable components. It divides a reuse re-engineering process into five sequential phases. Each phase is characterized by the object it produces. The first three phases are related to the identification, extraction and re-engineering of reusable components from existing software system, while the latter two phases are related to the

setting up of environments to support the reuse of the components considered in the previous phase. The RE^2 project does not aim to produce new software systems by reusing software components, but to produce reusable components from existing systems. For this reason, the phases emphasised the most in the paradigm are the first three.

Various candidature criteria looking for reusable assets in code have been described. Each of them has its own peculiarity based on the different kind of abstraction to be looked for, and each of them takes advantage of particular tools and structures to reach this aim. Tools based on the study of the dominance tree, the global variables, the user types of formal parameters in procedures, program slicing, testing and so on have been described.

Also the success of a candidature criteria depends also on the kind of software system it is applied. The quality of it can be measured on the basis of the number and the quality of the reusable software components identified. That quality is measurable by the use of quality attributes. Some quality attributes will be presented in the next chapter.

Chapter 3

New Method

3.1 Introduction

In the previous chapter a number of different methods looking for the extraction of reusable abstraction in existing software system have been investigated. Each of them follows different guide-lines and concepts to reach that aim. For the different characteristics of the methods the results that they provide, when applied to case studies, are quite different and categorize each of them as applicable to a particular kind of software system.

The study presented in this thesis has been centered particularly on the behaviour of two of the methods already introduced. The first is that defined by Canfora *et. al.* [22, 23] and based on the relationships existing between the user-defined data types and the procedure-like components (procedure or functions) that use them in their headings as formal parameters and/or a return type of functions. This method is called the ADT method. The second is that proposed by Cimitile *et. al.* [14] and based on the call graph and the dominance tree of the subject software system. The two methods have been presented as looking for two different kinds of abstractions. The ADT method looks for data abstraction, precisely abstract data types (ADTs); on the contrary, the dominance tree method looks for functional abstractions.

In reality, the new method can be interpreted as an extension of the ADT method motivated by the fact that the results that it supplied could be improved. In fact, the validity of a method for the extraction of valid modules that can be reusable depends on the quality of the method adopted. Five attributes can be used to define the quality of a method looking

for the identification of reusable modules from an existing system:

- *Adequacy*, measuring the capacity of the method to select those pieces of code that implement an abstraction;
- *Method Completeness*, measuring the property of the method to select all the pieces of code implementing an abstraction;
- *Purity*, measuring the capacity of the method not to include, in the selected pieces of code, data items and statements not belonging to the abstraction implemented;
- *Module Completeness*, expressing the property of the method to include, in the selected pieces of code, all the data items and the statements needed to implement the relative abstraction;
- *Identifiability*, indicating the ease of associating each piece of code selected with a concept within the application domain, that is with the abstraction it implements.

The first four attributes can be evaluated by using values like the size of the subject software system in terms of LOCs or number of software components (user-defined data types and procedure-like components), and the set of the candidate modules and their dimension in terms of LOCs or software components selected in each module.

Let S be the set of candidate modules extracted from the subject software system and S', a sub-set of S, the set containing only those candidate modules identified as reusable. The *adequacy* can be calculated, as the percentage of the number of the reusable candidate modules in relation to the number of those extracted by the following formula:

$$Adequacy = (\#S'/\#S) * 100,$$

where $\#S$ indicates the *cardinality* of the set S. Clearly, the major is the value of the *adequacy* the better adequate is the method to extract valid reusable modules.

Analogously, the *method completeness* can be measured as the percentage of the number of selected candidate modules with respect to the number of all the modules existing in the subject software system that can be proposed for reuse. The *Purity* can be obtained by the percentage of LOC in the candidate module that are copied into the final re-engineered module. The *module completeness* can be measured as the percentage of the pieces of code, either

as groups of lines of code or as software components, selected by an extracting-component method as implementing an abstraction with respect to the number of them necessary to implement that abstraction.

The *identifiability* attribute can be obtained as a percentage of values obtained in a particular case study. The value that is obtained does not depend just on the method used but it is influenced by how the application domain is complicated and elaborated.

In the next section the values of these quality attributes for the ADT method, and, the new method will be shown. A formal definition of the new method will be given in the successive paragraph.

3.2 The New Method

The starting point of the new method is the ADT method. This method has been used in experiments on four Pascal programs and the results obtained were quite satisfying. In fact, all the analysed programs were divided into sets of user-defined data type and procedure-like components, each of which could have implemented an abstract data type [23]. Problems arose in some case studies where there were both modules that were quite easy to understand and give meaning and semantics to, and modules that were quite large and very difficult to associate with an ADT. The evaluation of the quality attributes, introduced above, gave the following results:

Method Completeness	Adequacy	Purity	Module Completeness
50%-60%	40%-80%	50% - 100%	40%-70%

The attribute of identifiability has not been shown because its value is strictly associated with each case study. In the results shown above, a range of percentage for each attribute is given, showing the worst and the best results obtained. The lower values are justified by the fact that, in many cases, low quality groups of software components were produced, or even more than one group of software components were clusterized within the same candidate module. The first aspect influenced the lower limit in the estimate of the module completeness; whilst

the second aspect caused the lower values of all the other attributes. In fact, a big module clusterizing simpler modules, hid modules that could be candidate and associated to reusable ADTs, causing low values in the attribute of method completeness. On the other hand, it represented module not associable to reusable ADTs, influencing the adequacy. Sometimes it was possible to identify the ADT that could be implemented, this result was obtained by discarding some of software components selected, giving a low value for the attribute of purity.

Indeed, the observation of the experimental results and the modules candidate obtained revealed that the use of user-defined data types from numerous procedure-like components, belonging to different potential candidate ADTs, created a link between the subgraphs corresponding to each module thus causing more than one potential ADTs to be identified as a unique one. The result was modules that were quite large and very difficult to associate with ADTs.

The experimental results obtained demonstrated that the problem was usually caused by the user-defined data types that were sub-range or enumeration types and that were used, for example, to codify the possible states at the end of a number of different operations, or to define the definition range of indexes to arrays. The general semantic of that type motivated the massive use of them in a large number of procedure-like components. It appears that in the first case the information about the system loses meaning when the system is split into more than one sub-system; in the other cases, the definition ranges of the indexes can be codified using simple data types that are not considered in the identification of reusable abstract data types. The isolation of these kinds of types when they are selected to form the structure of an ADT together with more than one user-defined data types often leads to the splitting of a complex module into different modules that are simpler and for which it is possible to understand the meaning, and to associate a semantic and assume a re-engineering process. With the identification of more simpler modules this process brings to better values for the quality attributes of adequacy and method completeness. Clearly, in the re-engineering process it is necessary to take into consideration the isolated types, and it can be opportune to define one or more modules that define and export them, or to define them in the ADTs that use them.

Another peculiarity of the method above causing low value in the attribute of module completeness is that the candidate module is formed on the basis of the references of the procedure-like components to the user-defined data types. At the same time, this is a limitation of the method since the procedure-like components selected can call other procedure-like

components that do not reference the same user-defined data types, and that, for this reason, cannot be selected in some module. This causes a low value for the percentage of the number of the software components selected as implementing an ADT in respect to the number of them necessary to implement it and require human intervention to locate and add them to the set of software components that have been selected. To overcome this problem, as illustrated in [35], the use of the **dominance tree** and of the SDDT can help in the selection of these procedure-like components and in the identification of the relations existing both between them and the procedure-like components belonging to some modules and between the various modules. The starting point for this analysis is that, in the SDDT, the procedure-like components that do not belong to some candidate module and that are strongly dominated from the MAIN program are called only from the MAIN program, and they cannot be called from other procedure-like components belonging to some other module. By deleting them in the call graph it is possible to construct a new call graph that will no longer represent the original program but will still contain all the procedure-like components belonging to the modules and at least all the procedure-like components they call. By repeating the above process it is possible to eliminate from the call graph, and then from the SDDT, all the procedure-like components that in no way contribute to the implementation of an operation for some module. It is the last SDDT, so obtained, that is constituted exclusively by procedure-like components 'interesting' to define the reusable ADTs and to reveal the resources that they use for their implementation.

Another reason that brought to an improvement in results was that, for the rising complexity of the next phase of re-engineering, the ADT method was experimented with by considering only the procedure-like components declared in the MAIN program . In the next step the criterion was applied to the software components declared in some of the procedure-like component selected as one of the operation of a candidate abstract data type in order to obtain reusable modules that implemented abstract data types at different levels of abstraction. In the new method all the procedure-like components will be considered and the level of nesting will not be considered.

All these observations led to the definition of an algorithm whose starting point is the application of the ADT method for a first definition of the potential ADTs and their operators, and, next, by using the SDDT, continues with an iterative process using the SDDT to the identify all the procedure-like components not useful. The algorithm, articulated in seven steps, follows:

Step 1.

Apply the method for the extraction of the ADT.
If there exists some modules that are quite large and difficult to assign a meaning to and their supporting structures include subrange or enumeration types, then isolate them and reapply the criterion.
Label the obtained candidate modules

Step 2.

Construct the CDG of the program and apply the first rule of the method based on dominance tree to the identification of strongly connected subgraphs. Extract these subgraphs and transform the CDG to a CDAG. If the extracted subgraphs contain some nodes representing some procedure-like components belonging to some modules then the set of procedure-like components associated with the subgraph contribute to implement one or more functionalities of the associated ADTs

Step 3.

Construct the DDT and the SDDT of the CDAG and, for each of the above obtained modules, label with its own label all the nodes representing procedure-like components belonging to it

Step 4.

Delete in the CDAG all the procedure-like components that do not belong to any candidate module and that are associated in the SDDT with nodes that are directly and strongly dominated from the node MAIN, that is, from nodes that are connected with node MAIN by a marked edge

Step 5.

In the CDAG assume the existence of a connection between the MAIN program and all the procedure-like components that, after the deletion, have no arc coming in because they are called only from the deleted procedure-like components

Step 6.

Reconstruct the SDDT of the partial CDAG obtained above, and repeat from **Step 4** until a SDDT without nodes directly and strongly dominated from the node MAIN and associated with procedure-like components not belonging to any candidate module is obtained.

Step 7.

Mark each node, not directly and strongly dominated, with the label of the candidate mod-

ules containing procedure-like components, calling the procedure-like component that the node represents, or, if it is called from procedure-like components not belonging to any module, with their name.

The last SDDT obtained contains all the information for the identification of all the procedure-like components that participate, together with those already selected, to the implementation of the various functionalities of the candidate modules.

Before giving some basic rules that guide the extraction of this information it is opportune to specify that in the rest of this chapter and in the next chapter the formalism is adopted that a module candidate to represent an Abstract Data Type distinguishes the *interface specification* from the *implementation*. The interface specification sets out the name of the types involved and the operations that act on those types, that is all the information that the module exports. The implementation defines the representation of the type, the local resources and the implementation of the operations on the types, that is, all the information that the module hides. Later in the thesis EXPORT indicates the first part and BODY the second.

The basic rules for extraction are as follow:

Rule 1 Each MET subtree having only nodes representing procedure-like components from one candidate module implements a functionality of that module. In the implementation of this functionality the subtree can show the declarative structure of the involved procedure-like components [33]. In this case only the main functionality will be exported, but, if it is required to export some sub-functionalities, the procedure-like components implementing them can be declared in the same level of the main one.

Rule 2 If a procedure-like component (a set of procedure-like components) not belonging to any candidate module is represented by a node (a MET subtree) directly and strongly dominated from a procedure-like component included in a candidate module, it will belong to this module and will be nested in the dominating procedure-like component. If it is a subtree MET, as above, it can show the declarative structure of the involved procedure-like components.

Rule 3 If a procedure-like component belonging to a module directly and strongly dominates a procedure-like component belonging to a different module there will be a relation of USES from the first to the second module. The called procedure-like component cannot be nested in other procedure-like components and it will be exported.

Rule 4 If a procedure-like component belonging to a module is not strongly and directly dominated, being called from more than one procedure-like component, and these procedure-like components belong to the same module, it will be declared before the calling procedure-like component.

Rule 5 If a procedure-like component belonging to a module is not strongly and directly dominated, being called from more than one procedure-like component, and these procedure-like components belong to one or more different modules, there will be a relation of USE from the calling 'modules' to the called 'module'. The called procedure-like component cannot be nested in other procedure-like components and will be exported.

Rule 6 If a procedure-like component not belonging to a module is not strongly and directly dominated and is called from more than one procedure-like component belonging to one or more modules, it can be declared in each referencing module. The declaration of the procedure-like component will be repeated as many times as the number of the referencing modules, and if, in a module, only one procedure-like component calling it exists, it will be nested in the last one, otherwise its declaration will be in the same level of nesting and it will precede those of the calling procedure-like components.

Rule 7 If a procedure-like component not belonging to a module is called from one or more procedure-like components not selected to belong to a module but declared in some module on the basis of the **Rule 6**, then it will be declared in the same module.

Rule 8 If a procedure-like component belonging to a module is called from one or more procedure-like components not selected to belong to a module but declared in a module on the basis of the **Rules 6 and 7**, a relation of USE from the module to which the first procedure-like component belongs and the modules to which the last procedure-like components belong.

The above rules are general rules and if some case is not included in them they can be combined to cater for this case.

3.3 Formalization of the New Method

To permit automation of the new method it is necessary to provide a formal analysis of it. This can be achieved by expressing the algorithm and the set of rules introduced above by

the construction of a set of sets and relations. The notation adopted to express the sets and the relations are given in appendix A.

3.3.1 Abstract Data Types

Given a software system, if PP is the set of procedure-like components, and TT the set of user defined data types, then the following two relations can be defined:

- *used_to_define* expresses the use relation existing between the user defined data types. It is made up from TTxTT and is defined as:

$$used_to_define = \{t_i, t_j : TT | \textit{the user data type } t_i \textit{ is used to define } t_j \bullet (t_i, t_j)\};$$

- *proc_use_type* connects procedure-like components to the user defined data types that they use in the interface to define a formal parameter and/or a return type. It is made up from PPxTT and is defined as:

$$proc_use_type = \{p_i : PP, t_j : TT \mid p_i \textit{ uses } t_j \textit{ in its heading } \bullet (p_i, t_j)\}.$$

A **type-procedure-connection graph** (TPCG) is represented by the couple (N, E), where:

$$N \equiv PP \cup TT$$
$$E \equiv used_to_define \cup proc_use_type,$$

and expresses all the use relations existing between all the software components, and obtained from a static analysis of the software system. These relations constitute a subset of the set of direct relations of the system, that is those relations that summarise the meaningful reletionships existing among the software components. The set of direct relations constitutes the model to apply the candidature criterion. The latter consists on a set of summary relations obtained by combining the direct relations in expressions.
Starting from the TPCG a subgraph can be obtained that represents a first important step in the process of identification of abstract data types. Indicate with $\mu(n_i, n_j)$, with n_i and n_j two nodes of TPCG, a path [1] in TPCG, and with $\wp(TPCG)$ all the paths in TPCG, then the subgraph, indicated with TPCG$' \equiv (N', E')$ expresses a subset of the summary relations and can be obtained by the following construction:

[1] A path of a graph $G = (E, N)$ is a sequence of nodes $n_1, n_2, ..., n_k$, belonging to N, such that every couple (n_i, n_{i+1}) belongs to E.

$N' = N$
$E' = E - \{p_i : PP, t_j : TT \mid \exists\ t_k : TT \bullet (proc_use_type(c_i, t_k) \wedge$
$\exists\ \mu(t_j, t_k) \in \wp(TPCG)) \bullet (p_i, t_j)\} - \{t_i, t_j : TT \mid used_to_define(t_i, t_j) \bullet (t_i, t_j)\}$

Practically, in the subgraph so obtained, the only connections existing are between the procedure-like components and user defined data types, on the basis of the following property: a procedure-like components is connected to a user defined data type only if the former does not use as formal parameter a user defined data type using the latter for its definition.
Each isolated sub-graph[2] in TPCG can represent the basis for the construction of abstract data types.

3.3.2 Dominance Tree

The **call directed graph** (CDG) defining a software system can be defined as the triple (s, PP, EE), in which s is the MAIN program, PP is the set of all the procedure-like components of the subject software system, then $\{s\} \cup PP$ is the set of nodes, and EE, made up from $\{s\} \cup PP$ x PP and defined by:

$$EE \equiv call = \{x : \{s\} \cup PP, y : PP \mid the\ component\ x\ calls\ the\ component\ y\}$$

represents the *call* relations existing between the procedure-like components of the program. The *call* relation is another kind of direct relation.
The CDG of a program can be generated automatically from the code and from this can be derived the **call directed acyclic graph** (CDAG) by collapsing every strongly connected subgraph[3] into one node [16]. The CDAG is defined by the triple (s, PP', EE'), where each procedure-like component in PP' is an element of PP or a collapsed subgraph of CDG, and EE' is defined by the same call relation of EE, with the difference that this time the involved procedure-like components are those of PP'.
Supposed that the CDG is reduced to its CDAG, then the triple (s, PP, EE) will be used to represent the CDAG; moreover by representing with $\mu(x, y)$ a path connecting x and y in the CDAG, the following can be defined:

- the *dominance relation* is a relation on $\{s\} \cup PP$ x PP, defined by:

[2] An isolated sub-graph of a graph G is a graph g such that: $g \subset G \wedge g \neq \emptyset \wedge \exists \neg g \subset G \bullet (\neg g \neq \emptyset \wedge g \cap \neg g = \emptyset \wedge g \cup \neg g = G)$, where the symbol $\emptyset$ denotes the empty graph and $\neg g$ indicates the subgraph complement of the subgraph g respect the graph G. This definition is due to Calliss [34].

[3] A strongly connected subgraph, belonging to the graph CDG, is defined as the couple (PP_i, EE_i) such that: $PP_i \subset PP \wedge EE_i \subset EE \wedge (\forall\ p, q : PP_i\ \exists\ \mu(p, q) \wedge \mu(q, p))$

$$dom = \{x : \{s\} \cup PP, y : PP \mid \forall \mu(s,y) \bullet x \in \mu(s,y) \bullet (x,y)\},$$

then $dom(x,y)$, that is x *dominates* y, if and only if every path from the initial node of the graph to y span x;

- *direct dominance relation* is a relation on $\{s\} \cup PP$ x PP, defined by:

$$dir_dom = \{x : \{s\} \cup PP, y : PP \mid dom(x,y) \wedge (\forall z : PP \bullet (z \neq x) \wedge (z \neq y) \wedge dom(z,y) \Rightarrow dom(z,x)) \bullet (x,y)\},$$

then $dir_dom(x,y)$, that is x *directly dominates* y,if and only if x dominates y and each node, different from x and different from y, that dominates y dominates x; in other words x is the nearest node to y between all the nodes that dominate y;

- *strongly direct dominance relation* is a relation on $\{s\} \cup PP$ x PP, defined by:

$$str_dir_dom = \{x : \{s\} \cup PP, y : PP \mid dir_dom(x,y) \wedge call(x,y) \wedge (\nexists z : PP \bullet (z \neq x) \wedge call(z,y)) \bullet (x,y)\},$$

then $str_dir_dom(x,y)$, that is x ***strongly direct dominates*** y, if and only if x directly dominates y and is the only node that calls y;

- the **direct dominance tree** (DDT) of CDAG is a tree defined as a triple (s, PP, ED), where $ED \equiv dir_dom$;

- the **strongly direct dominance tree** (SDDT) of CDAG can be obtained from the DDT by marking all the edge corresponding to the procedure-like components of the relation *str_dir_dom*. The set of all subtrees of a SDDT is composed of:

  - the set MET of all subtrees having only marked edges, and
  - the set UMET of all subtrees having both marked and unmarked edges.

3.3.3 Formalization

The application of the ADT method gives a first approximation of the modules that can be proposed for reuse, and a first identification of the set of software components that will constituted each module. Indicate with MOD the set of candidate modules obtained, then two relations of '*module composition*' between the identified modules and the software components can be identified. The first relation, ***mod_proc_comp***, is defined on MODxPP, and expresses for each module the procedure-like components that it includes:

$mod_proc_comp = \{m : MOD, c : PP \mid the\ procedure-like\ component\ c\ belongs\ to\ the\ module\ m \bullet (m, c)\};$

the second, *mod_type_comp* , is defined on MODxTT, and for each module identifies the user-defined data types that contribute to it:

$mod_type_comp = \{m : MOD, t : TT \mid the\ user-defined\ data\ type\ t\ belongs\ to\ the\ module\ m \bullet (m, t)\}.$

The relation will be enriched during the process as other new procedure-like components belonging to the modules are discovered. Once the first classification is established then the observation of the CDG offers initial information about the relations existing between procedure-like components both belonging and not belonging to the modules, and also about the relations between modules. In fact, the existence of strongly connected subgraphs in the CDG indicates a cooperation between the procedure-like components associated with the nodes of each subgraph for the implementation of one or more functionalities. These functionalities are associable to the modules candidate including the procedure-like components. If the modules involved are more than one the strongly connected subgraphs indicate also a cooperation between these modules, and, then, a USE relation existing between them.

The process continues with the construction of the CDAG and with the identification of the relations *dom*, *dir_dom* and *str_dir_dom*.

The survey of all the potential reusable procedure-like components can be carried out using the following iterative process. Starting with the labelling of all the elements (sets, relations, and so on) known at this stage with the number 0, the notation adopted marks all the analogous elements obtained in each iteration with a progressive subscript. Moreover, $call_G$, dir_dom_G and $str_dir_dom_G$ indicate respectively the call connections between nodes, the *direct dominance* and the *strongly direct dominance* relations referred to the particular graph G. Indicate with $CDAG_0 = (s, PP_0, EE_0)$ the call directed acyclic graph with the relative sets of nodes and arcs, the iterative process will finish when the following position is reached:

$$\exists n : N \bullet P_{n+1} = \{c : PP_n \mid str_dir_dom_{CDAG_n}(s, c) \wedge \nexists\, m : MOD \bullet mod_proc_comp(m, c)\} = \emptyset.$$

P_{n+1} represents the set of the nodes of a reduced call directed acyclic graph, $CDAG_n$, that can be reached with the characteristic that if some procedure-like components strongly direct dominated from the MAIN program exist, then they belong to some modules. The graph $CDAG_n$ contains all the interesting procedure-like components, that is, only those

participating in the implementation of the modules and in no other.

The following exemplify the iterative process that leads to the attainment of the position above:

evaluate $\forall k : 1 ... n \quad CDAG_k = (s, PP_k, EE_k)$ where:

$$P_k = \{c : PP_{k-1} \mid str_dir_dom_{CDAG_{k-1}}(s, c) \wedge \nexists\, m : MOD \bullet mod_proc_comp(m, c) \bullet c\}$$

$$PP_k = PP_{k-1} - P_k,$$

and:

$$E_k = EE_{k-1} - \{c : P_k \bullet (s, c)\} - \{c : P_k, d : PP_{k-1} \mid call_{CDAG_{k-1}}(c, d) \bullet (c, d)\}$$

$$EE_k \equiv call_{CDAG_k} = E_k \cup \{c : PP_k \mid \nexists\, d \in PP_k \bullet (d, c) \in E_k \bullet (s, c)\},$$

that is the set of nodes PP_k of $CDAG_k$ is obtained from the set of nodes PP_{k-1} of $CDAG_{k-1}$ by deleting in the latter the set P_k of the nodes corresponding to procedure-like components strongly direct dominated from the MAIN program and for which modules containing them do not exist. The definition of EE_k can be achieved in two steps: deletion from EE_{k-1} of all the couple of procedure-like components representing arcs coming in and coming out from the deleted nodes; addition to the set E_k obtained above of couples of procedure-like components representing assumed call connections between the MAIN program and all the nodes representing procedure-like components that, after the deletion, have no arc coming in because the associated procedure-like components were called only from the deleted procedure-like components.

After the construction of the $CDAG_k$, determine $dir_dom_{CDAG_k}$ and $str_dir_dom_{CDAG_k}$.

At the end of the iterative process the set PP_n contains all the procedure-like components that are essential for the implementation of the ADTs identified by the ADT method. The procedure-like components that are strongly direct dominated are called only from the dominating procedure-like components; on the contrary, the call relations are more complex for the procedure-like components not strongly direct dominated. This set can be defined by the following:

$$DDC = PP_n - \{c : PP_n \mid \exists d : \{s\} \cup PP_n \bullet str_dir_dom_{CDAG_n}(d, c) \bullet c\}.$$

It shows all the procedure-like components that are *directly dominated* but not *strongly direct*

dominated.

The relation, *calling direct dominated procedure-like component* (*call_dir_dom_comp*) defined on $(MOD \cup PP_n) \times DDC$, expresses for each procedure-like component in DDC if it is called from procedure-like components belonging to some modules, including, in this case, the couple that has as first element the label of the module and as second element the subject procedure-like component, or if it is called from a procedure-like component whose membership to some module is not yet established. In the last case, the couples enriching the relation are composed from the calling procedure-like component as first element and the subject procedure-like component as second element. The formal definition is as follows:

$$call_dir_dom_comp = \{m : MOD, c : DDC \mid \exists d : PP_n \bullet mod_proc_comp(m,d) \land call_{CDAG_n}(d,c) \bullet (m,c)\} \cup \{d : PP_n, c : DDC \mid call_{CDAG_n}(d,c) \land \nexists\, m : MOD \bullet mod_proc_comp(m,d) \bullet (d,c)\}$$

The relation obtained together with $str_dir_dom_{CDAG_n}$ and $call_{CDAG_n}$ contains all the information for the identification of the relations and the belongings to modules of all the procedure-like components, whose membership to modules have not been established by the ADT method and that participate, together with those already selected, to the implementation of the various functionalities of the candidate modules.

Other useful relations that will be considered to define the re-engineering process for the survey of the relation cited above, are defined as:

- *module defines procedure-like components* (*mod_def_proc*), defined on $MOD \times PP_n$, indicates for each module all the procedure-like components it defines:

  $mod_def_proc = \{m : MOD, c : PP_n \mid module\ m\ defines\ the\ component\ c \bullet (m,c)\};$

- *module exports procedure-like components* (*mod_exp_proc*), defined on $MOD \times PP_n$, indicates for each module all the components it exports:

  $mod_exp_proc = \{(m,c) : mod_proc_comp \mid module\ m\ exports\ component\ c \bullet (m,c)\},$

  this definition is justified from the fact that each module will export at most the procedure-like components that belong to some module as defined in the relation *mod_proc_comp* , and it is revealed in the application of the method based on the use of user-defined data types in the interface. In fact, that method is useful to identify the operators of the candidate ADTs, that is, those procedure-like components using

in their interface the user-defined data types on which the structures of the ADTs are constructed;

- *procedure-like component declares procedure-like component* (*proc_dec_proc*), defined on $PP_n \text{x} PP_n$, indicates the nesting of procedure-like components:

  $$proc_dec_proc = \{c, d : PP_n \mid \exists\, m : MOD \bullet mod_def_proc(m, c) \wedge mod_def_proc(m, d) \wedge d \text{ is nested in } c \bullet (c, d)\},$$

  then $proc_dec_proc(c, d)$ is true if and only if the procedure-like components c and d belong to the same module m and the former procedure-like component declares the latter;

- *module USES module* (*mod_uses*), defined on MOD x MOD, indicates the relation of USE between modules:

  $$mod_uses = \{l, m : MOD \mid (\exists\, c : PP_n \bullet mod_def_proc(l, c)) \wedge (\exists\, d : PP_n \bullet mod_exp_proc(m, d)) \wedge call_{CDAG_n}(c, d) \bullet (l, m)\},$$

  that is $mod_uses(l, m)$ is true if and only if a procedure-like components belonging to module l exists and it calls a procedure-like component belonging and exported from module m.

The relation *mod_def_proc* initially coincides with the relation *mod_proc_comp* because the membership established in the first step by the ADT method continues to be valid. With an iterative process its structure is enriched until one couple exists for each element in PP_n. To define the process the new relation *module calls procedure-like component* (*mod_call_proc*) obtained by restricting the domain of the relation *call_dir_dom_comp* to MOD, is proposed:

$$mod_call_proc = MOD \lhd call_dir_dom_comp.$$

Then, established the initial condition in the following:

$$mod_def_proc = mod_proc_comp,$$

and supposed that the final condition of the iterative process is:

$$rng(mod_def_proc) = PP_n,$$

that is the membership for all the procedure-like components are established, the enrichment process can be exemplified in the following steps:

$mod_def_proc' = mod_def_proc \ \cup \ \{m : MOD, c : PP_n \mid$

$\nexists\, l : MOD \bullet mod_proc_comp(l, \ldots$

$(mod_call_proc(m, c) \vee (\exists\, d : PP_n \bullet mod_def_proc(m, d) \wedge str_dir_dom_{CDAG_n}(d, c)))$

$mod_def_proc = mod_def_proc'.$

$mod_call_proc = \{m : MOD, c : DDC \mid \exists d : PP_n \circ mod_def_proc(m, d) \wedge$

$call_{CDAG_n}(d, c) \bullet (m, \ldots$

The first step of the iterative process adds new terms to the ***mod_def_proc*** relatic procedure-like components not belonging to any module. For each new term, tl element is the name of the processed procedure-like component, while the first bel of the module that contains a procedure-like component that, with referen $CDAG_n$, strongly direct dominates or simply calls the subject procedure-like co The last information is revealed from the ***mod_call_proc*** that is updated as new me of calling procedure-like components is established. In fact, as the number of p like components belonging to modules increases it is possible to define the calli direct dominated procedure-like components in terms of modules and no longer ir procedure-like components without belonging to modules. Clearly, in each itera the relation *mod_def_proc* is also updated.

The definition of the relation *mod_exp_proc* follows from the next consideration. Ea MET (a subtree such that its edges express exclusively strongly direct dominan SDDT indicates a functionality implemented in the program [14]. Because the p like components belong to a module the functionality identified from a MET, in re functionality of the module to which the procedure-like component associated witl of the MET subtree belongs. Since it is a functionality of a module and being used f modules it has to be exported from the module to which it belongs. From these obs and considering that the procedure-like components that can be exported from are only those whose membership to that module has been established by the a of the ADT method the relation *mod_exp_proc*, defined on MOD x PP_n, is sp the following, where it is used the relation *mod_proc_call*, defined by the applicati iterative process illustrated above:

$mod_exp_proc = \{m : MOD, c : PP_n \mid mod_proc_call(m, c) \wedge$

$\nexists\, d : PP_n \bullet (mod_def_proc(m, d) \wedge str_dir_dom(d, c)) \bullet (\ldots$

The above relation expresses that, in each MET subtree, only the procedure-like components belonging to some module and associated with nodes of the subtree that are not strongly direct dominated from procedure-like components belonging to the same module are exported, that is declared in the *interface specification* of the module. All the procedure-like components associated with nodes that are strongly direct dominated from nodes representing procedure-like components belonging to the same module are used only from procedure-like components from the same module and not from procedure-like components from different modules, and, thus are considered subfunctionalities of a *main* functionality of the module and declared in the *implementation* of the module. It can happen that some of those subfunctionalities associated with some of the internal node in the subtree whose procedure-like component associated is not exportable from the rule given above are interesting to complete the set of operations that a module can export and to be declared in the *interface specification* of the module. This choice can be evaluated by the software engineer, and the relation *mod_exp_proc* can include other couples.

Once the membership of the modules is defined for all the procedure-like components and it is established which of them are exported from the various modules the next step is to define how the procedure-like components can be nested in the *implementation* section of the modules. That is, to define the *proc_dec_proc* relation. Clearly, if a procedure-like component is exported from a module it cannot be nested in some other procedure-like component. For the purpose of defining the nesting the *call* and *strongly direct dominance* relations are very important. In fact, if two procedure-like components belong to the same module and a strongly direct dominance relation exists between them then the dominating procedure-like component is the only one to call the other. For this reason if the latter procedure-like component is not chosen to be exported if can be declared in the former. Analogously, if a procedure-like component whose membership of a module has not been established on the basis of the use of user-defined data types in the interface but, in the next definition of the *mod_def_proc*, is called from only one procedure-like component of the module it belongs to it can be nested in the calling procedure-like component. The translation of the above assertions in the formal notation adopted brings to the definition of the relation *proc_dec_proc* that follows:

$$proc_dec_proc = \{c, d : PP_n \mid \exists\, m : MOD \bullet mod_def_proc(m, c) \wedge mod_def_proc(m, d) \wedge \neg mod_exp_proc(m, d) \wedge (str_dir_dom_{CDAG_n}(c, d) \vee ((\exists l : MOD \bullet mod_def_comp(l, d)) \wedge (\exists!\, e : PP_n \bullet mod_def_proc(m, e) \wedge call_{CDAG_n}(e, d) \wedge e = c)) \bullet (c, d)\}$$

The final step is to define how the identified modules use each other. In fact, the procedure-

like components belonging to a module *call* or ***strongly direct dominated*** procedure-like components belonging to other modules. These ***call*** and ***dominance*** relations establish a USE relation between the modules. If a module includes a procedure-like component that strongly direct dominates a procedure-like component belonging to another module then the former module USEs the latter. Obviously, the same USE relation is established between a module containing procedure-like components calling other procedure-like components included in other modules. The relation *mod_uses* is defined as follows:

$$mod_uses = \{l, m : MOD \mid (\exists c : PP_n \bullet mod_exp_proc(m, c)) \wedge (mod_call_proc(l, c) \vee (\exists d : PP_n \bullet mod_def_proc(l, d)) \wedge str_dir_dom_{CDAG_n}(d, c)) \bullet (l, m)\}.$$

Until now, the structure of the modules has been discussed exclusively in terms of procedure-like components while the user-defined data types have been neglected. It is now opportune to consider them because they constitute the supporting structure of the candidate ADT.

The relation *used_to_define* connects couples of user-defined data types according to if a user-defined data type is used to define another user-defined data type. On the contrary, the relation *mod_type_comp* connects modules to the user-defined data types contained in them. The last relation has been introduced after the presentation of the method based on the use of user-defined data types in the interfaces of the procedure-like components. The information obtained by using that method are only partial because the membership of modules is not defined for some of the types and precisely for some of the sub-types[4] of the types already selected.

A process analogous to that illustrated above for the procedure-like components can be used for the identification of the interesting user-defined data types and their membership to modules. Also, the relations *module defines type* (*mod_def_type*) over MODxTT indicating which user-defined data types are defined in each module, and *module exports type* (*mod_exp_type*) over MODxTT indicating which user-defined data types are exported from each modules, are defined. In reality, the second relation, *mod_exp_type*, coincides with the relation *mod_type_comp*. In fact, all the user-defined data types identified in the first step of the entire process are exportable from the modules because they are referenced from the the procedure-like components exported from the modules. The relation *mod_def_type* can be identified by using an iterative process similar to that used for the construction of the relation *mod_def*. The starting point for the iterative process is:

[4]Liu and Wilde say that the user-defined data type t_1 is a subtype of the user-defined data type t_2 if a path exists that connects t_1 and t_2. In that case t_2 is a super-type of t_1

$mod_def_type = mod_type_comp,$

since the relation *mod_type_comp* is the initial part of the searched relation. The iterative process adds couples from MODxTT to the relation according to if a type belonging to some module uses, for its definition, other types that are not already defined in the same module or exported from other modules. Thus established the iterative process stops when types that are not defined in the same module of the types that use them for their definition and not exported from other modules do not exists. Formally translated, the final condition is:

$$\{t : TT \mid \exists\, r : TT \bullet used_to_define(t,r) \;\wedge\; \exists\, m : MOD \bullet mod_def_type(m,r) \;\wedge\; \neg mod_def_type(m,t) \;\wedge\; \nexists\, l : MOD \;\bullet\; mod_exp_type(l,t)\} = \emptyset,$$

the iterative process is exemplified in the following:

$$mod_def_type' = mod_def_type \;\cup\; \{m : MOD, t; TT \mid \nexists\, l : MOD \;\bullet\; mod_exp_type(l,t) \;\wedge\; \exists\, r : TT \bullet used_to_define(t,r) \;\wedge mod_def_type(m,r) \;\bullet (m,t)\}.$$

$mod_def_type = mod_def_type'$

Clearly, the relation *mod_uses* is enriched according to how some user-defined data types exported from some modules are used to define user-defined data types belonging to other modules. On the basis of that the relation *mod_uses* is completed according to the following rules:

$$mod_uses' = mod_uses \;\cup\; \{l, m : MOD \mid \exists t : TT \bullet mod_exp_type(m,t) \;\wedge\; \exists\, r : TT \bullet\; mod_def_type(l,r) \;\wedge\; used_to_define(t,r) \bullet (l,m)\}$$

$mod_uses = mod_uses'.$

3.4 Summary

In this chapter by using an discursive approach is proposed a method to improve the reusability of ADTs extracted from code. The same method has been presented by using a formal definition of all the relations existing between the various components identified. The formalization makes easier to automate the proposed method.

The identification of ADTs is based on the relation of *use* existing both between user-defined data types, and between procedure-like components and user-defined data types. A very large and complex module can be split into more than one simple module by isolating a particular kind of user-defined data type.

The method to restructure each ADT is based on the Dominance Tree and the Strong Direct Dominance Tree both obtained from the call graph of the system. By observing the SDDT all the procedure-like components that are not called from the procedure-like components belonging to some ADT are identified and, by reiterating the algorithm presented, the complete set of all software components used to implement some operation in some ADT is obtained.

Clearly the intervention of a software engineer continues to be of fundamental importance to assign a meaning to the obtained modules.

Chapter 4

Implementation

4.1 Introduction

In the third chapter the new method proposed was formulated as a set of sets and relations. This kind of representation has the added advantage of a direct way to implementation by using a logic programming language, for example Prolog. Then, in order easily to perform experiments to evaluate the approach proposed, a prototype tool that implements the method proposed has been developed.

The prototype is intended to be used in order to evaluate the case studies and not for commercial reasons. Therefore, issues like time/space performance and user-friendliness have not been taken into account. The tool takes into account two aspects that are very important for a tool that fits into a research environment, the aspects of flexibility and of easy to evolve. The importance of these two peculiarities derives from the high probability that the methods and the experimental environment can change. In addition, the prototype tool has to offer the possibility of defining new summary relations to be able to choose the level of abstractions to be looked for and the characteristics of the results expected. If the methods and the tools are versatile and easily tailorable they are easy to evolve on the basis of the new knowledge developed both in the environment in which they are used and in the research community.

The implementation of the approach requires three fundamental components:

a repository to store the direct relations;

a language to define the summary relations;

a query facility to specify the type of abstraction to be looked for.

Prolog puts at disposal all the instruments to supply those components. In fact a Prolog dictionary can be used to record the direct relations and the production rules can be used to define the summary relations, while Prolog queries can be used to express the abstraction to be looked for. Besides, the Prolog language offers the versatility cited above; in fact in a Prolog implementation it is possible to easily define new summary relations.

The expansion of the set of the summary relations is exactly what has happened with the prototype tool implementing the approach proposed. Starting from the prototype tool implementing the ADT method it has been possible to define all the new relations defined in the third chapter.

In the next section a description of the prototype tool implementing the ADT method is given, followed by the description of the expanded method.

4.2 ADT Method Implementation

One of the components of that implementation is a static code analyser written using Yacc [39], a standard Unix facilities for the implementation of a parser. The current version analyses Pascal program written according to the ISO standard and the main disadvantage is that the programs to be analysed have to be written in a unique compilable file. The analyser is used for automatically producing the program dictionary recording the direct relations. The program dictionary is composed as follows:

- facts of arity 1, define each software components involved in the candidature criterion and state their type. In each fact, the argument indicates the name of the subject software component, while the name of the fact states its type, that is if it is procedure, function or user-defined data type:

```
proc(procedure_name)
func(function_name)
user_def_type(type_name)
```

- facts of arity 2, indicating the kind of relation involved between a couple of software components. In each of these facts, the arguments indicate the names of two software components involved while the name of the fact states the type of relation between the cited components:

```
proc_use_type_in_interface(procedure_name,type_name)
func_use_type_in_interface(procedure_name,type_name)
used_to_define(type_name_1,type_name_2)
proc_func_dec(procedure_name_1,procedure_name_2)
proc_func_call(procedure_name_1,procedure_name_2)
```

  The first two facts state that **procedure_name** is a procedure/function that uses in its interface **type_name** to declare a formal parameter, The third fact indicates that **type_name_1** is used in the definition of **type_name_2**, the fourth that the procedure/function **procedure_name_1** declares the procedure/function **procedure_name_2** and the last that the procedure/function **procedure_name_1** calls procedure/function **procedure_name_2**. And a fact exists for each couple of software components between which one of the relations defined above exists.

Once that the program dictionary has been produced the summary relations implementing the candidature criterion can be easily computed. Production rules computing them have been implemented as well as programs to simplify the interrogation of the system. In particular the programs have been written to answer to the following query:

- **adt_strutt(T,T_set)**, that gives the set **T_set** of user-defined data types that belong to a cobweb of formal parameters declaration around the type **T**. All this user-defined data types are candidate to implement the supporting data structure of a candidate ADT;
- **adt_op(T,P_set)**, that returns the set **P_set** of the procedure-like components that use at least one of the user-defined data type belonging to the set **T_set**, that is to the coweb of user-defined data types constructed around **T**. This set of procedure-like components will define the operations of the candidate ADT whose data structure has been defined by the coweb of user-defined data types constructed around **T**.

These two queries are combined in the unique query **adt(T,T_set,P_set)** that gives both **T_set** and **P_set** that is the supporting structure and the operations of the candidate ADT constructed around the user-defined data type **T**.

```
vega(sun4):dcs3mt[30]:  swipl
Welcome to SWI-Prolog (Version 1.8.6 December 1993)
Copyright (c) 1993, University of Amsterdam.  All rights reserved.

1 ?- ['STARTUP'].
STARTUP compiled, 0.02 sec, 636 bytes.

Yes
2 ?- load.
adt_abstractor.prolog compiled, 0.03 sec, 3,136 bytes.
common_definitions.prolog compiled, 0.03 sec, 3,276 bytes.
utility.prolog compiled, 0.02 sec, 660 bytes.
adt_database.prolog compiled, 0.15 sec, 14,124 bytes.

Yes
3 ?- project_program_dictionary(main).

Yes
4 ?- adt(tracestring,T_set,P_set).

P_set = [etrace]
T_set = [tracestring]

Yes
5 ?- adt(lineptr,T_set,P_set).

P_set = [alloline,freeline,getind,getnew,getpak,gettxt,linkup]
T_set = [lineptr]

Yes
6 ?- adt(argstringstring,T_set,P_set).
P_set = [amatch,catsub,dumppat,getccl,getrhs,locate,makpat,maksub,match,omatch,patsiz,stclos,subst]
T_set = [argstring,patternstring]

7 ?- adt(filenamestring,T_set,P_set).

P_set = [assignfile,doread,dowrit,getfn,open]
T_set = [filenamestring]

Yes
8 ?- adt(lineptr,T_set,P_set).

P_set = [addset,ctoi,esc,filset,inject,readcmd,readline,readterm]
T_set = [linestring]

Yes
9 ?- adt(linelength,T_set,P_set).

No 10 ?- halt.
```

Table 4.1: Unix script for query ADT

Table 4.1 shows an execution of the prototype tool experimented with one of the case studies that will be presented in the next chapter.

For each query `adt(T,T_set,P_set)` the program answer 'Yes' with a result for *T_set* and *P_set* if the user-defined data type T can generate a candidate ADT otherwise it answers 'No' like for the user-defined data type *linelength.*

4.3 New Method Implementation

The implementation of the ADT method presented above has been extended to the production of all the summary relations introduced in the formal definition of the method. All the Prolog production rules have been grouped in different Prolog files on the basis of the kind of relations that they produce and of the algorithm that they implement.

A description of all the files with the facts that they produce follows.

`pm_comp.prolog` - by using the results from the implementation of the ADT method, produces the facts of arity 2 indicating the user-defined data types and the procedure-like components that constitute the supporting data structure and the set of operations for each module implementing a candidate ADT. Assigned an index to each module the facts produced are:

```
mod_type_comp(num_module,type_name)
mod_proc_comp(num_module,procedure_name)
```

They indicate that the user-defined data type **`type_name`** and the procedure-like component **`procedure_name`** have been established from the ADT method belonging to the candidate module **`num_module`**;

`pm_dom_proc.prolog` - calculates the *direct dominance relations* existing between the procedure-like components. The algorithm used is an iterative one. In each iteration it calculates the path with length progressively increasing starting from the path of length one. For each procedure-like component its dominating procedure-like component is found if a length of path is achieved such that all the paths of this length reaching the subject procedure-like component has one node in common. The procedure-like component corresponding to that node is the direct dominating component.

Consider a call direct acyclic graph $CDAG = (s, PP, EE)$, by using the the notation introduced in the Appendix A and indicating with *dir_dom* the relation defined $\{s\} \cup PP$ x PP and expressing the direct dominance the algorithm implemented in `pm_dom_proc.prolog` appears as follows:

Established that the initial conditions are:

$dir_dom = \emptyset \qquad k = 1$

and supposed that the final condition of the iterative process is:

$rng(dir_dom) = PP \ \vee \ k = \#PP$

that is a direct dominating procedure-like component has been found for each procedure-like component or k is equal to the number of the procedure-like components, the iterative process is exemplified as:

$$dir_dom' = dir_dom \cup \{x : \{s\} \cup PP, y : PP \mid$$
$$(\forall \mu(z,y) \in \wp(CDAG) \bullet l(\mu(z,y)) = k \ \wedge \ x \in \mu(z,y) \bullet (x,y)\}$$

$dir_dom = dir_dom'$

$k = k + 1$

The program produces a database of prolog facts of arity three as follows:

```
dom_proc(procedure_name_1,len_path,procedure_name_2).
```

The first argument of the fact indicates the direct dominating procedure-like component, the third the direct dominated procedure-like component and the second the length of the longer path in the CDAG that connect the two procedure-like components. Clearly if `len_path` is equal to one the relation existing between the two procedure-like components is a relation of strongly direct dominance;

`pm_algo_mod_proc_comp.prolog` - implements the iterative process looking for the full set of procedure-like components that cooperate to the implementation of ADTs extractable from the software system. The iterative process has been illustrated in the Section 3.3, and it performs a deletion of the procedure-like components not interesting in the implementation of the ADTs identified. This deletion involves a change in the database of the direct relations by deleting of the facts regarding the not interesting procedure-like components and adding new facts `proc_func_call(procedure_name_1,`

procedure_name_2). When executed the program gives information like:

```
nb_iteration_mod_proc_comp(number_iteration)
change(pm_algo_mod_proc_comp,proc_func_call)
```

indicating the number of iteration performed and the information that the da of facts proc_func_call(procedure_name_1,procedure_name_2) is changed;

pm_dir_dom_comp.prolog - identifies the set of procedure-like components, chosen b those considered interesting for the implementation of the ADTs, and that are dominated but not strongly direct dominated, that is it identifies the set DDC c in Chapter 3. It produces fact of arity 1:

```
dir_dom_comp(procedure_name)
```

indicating that the procedure-like component **procedure_name** is not strongly dominated;

pm_call_dir_dom_comp.prolog and pm_mod_call_proc.prolog - implement the re *call_dir_dom_comp* recognizing for all the procedure-like components not strongly dominated identified with the previous program if they are called from procedur components not belonging to modules or procedure-like components belonging to ules. The first produces facts of arity 2 as:

```
call_dir_dom_comp(procedure_name_1,procedure_name_2)
```

indicating that the first procedure-like component calls the second that is a not stro direct dominated procedure-like component. The second produces the following of facts of arity 2:

```
mod_call_proc(num_module,procedure_name)
```

indicating that the module **num_module** contains a procedure-like component ca **procedure_name**.

Those facts are used in the next program to identify the membership to module the procedure-like components whose membership to modules is not yet identified

pm_mod_def_proc.prolog and pm_mod_def_type.prolog - calculate the relations *mod_de* and *mod_def_type* respectively, that is the membership to modules of the software c

ponents for which it has not yet been defined, by implementing the two iterative processes illustrated in Section 3.3. They produce two sets of facts of arity 2 of the kind that follows:

```
mod_def_proc(num_module,procedure_name)
mod_def_type(num_module,type_name)
```

The first kind, produced from the first program, indicates that the module `num_module` contains the component `procedure_name`, while the second produced from the second program identifies the type `type_name` as belonging to the module `num_module`. The first program produces information like:

```
nb_iteration_mod_def_proc(number_iteration)
change(pm_mod_def_proc,mod_call_proc)
```

indicating the number of iterations performed and that the database of facts *mod_call_proc* has been changed. The second program produces information like:

```
nb_iteration_mod_def_type(number_iteration)
change(pm_mod_def_type,mod_uses)
```

indicating the number of iteration executed and that the database *mod_uses* indicating the USE relation between the modules is changed;

`pm_mod_exp_proc.prolog` and `pm_mod_exp_type.prolog` - define the relations *mod_exp_proc* and *mod_exp_type* respectively, expressing software components exported from each modules, that is the interface specifications. The facts they produce are of arity 2 and are of the following type:

```
mod_exp_proc(num_module,procedure_name)
mod_exp_type(num_module,type_name)
```

The first kind of fact is produced by the former program while the second by the latter. They express that the module `num_module` exports the software components `procedure_name` and `type_name` respectively.

pm_proc_dec_proc.prolog - implements the relation *proc_dec_proc* indicating the nesting of procedures and produces facts of arity 2 as follows:

```
proc_dec_proc(procedure_name_1,procedure_name_2)
```

expressing that the procedure-like component procedure_name_2 is nested in procedure_name_1

pm_mod_uses.prolog - defines the relation of use existing between modules and the facts are of arity 2 as:

```
mod_uses(num_module_1,num_module_2)
```

where num_module_1 uses num_module_2.

All the programs illustrated above can be executed either automatically all in the same time or one at a time. Because the definition of some kinds of facts depends on the definition of other kind of facts, a dependency exists between the programs. The execution of some programs has to follow the execution of other programs and, if it has been decided to evaluate one program at a time, it is possible to query the system to find out which programs has to be executed first. To obtain this information it possible to use the query pm_FACT_dependency(true) where pm_FACT indicates one of the program introduced.

Table 4.2 shows a use of dependency query for the predicate *pm_mod_def_proc*. The execution of this predicate depends on the definition of *mod_proc_comp* produced by *pm_comp*, on the execution of *pm_algo_mod_proc_comp* that depends on the definition of *mod_proc_comp* and *dom_proc*, and on the definition of other predicates as the Unix script indicates. The execution of the predicate *pm_comp*, *pm_dom_proc* and *pm_algo_mod_proc_comp* reduces the dependencies of *pm_mod_def_proc* from other definitions.

The set of facts produced by one the program defined can be saved in a file by using the predicate pm_FACT_save('file_name'), where *pm_FACT* indicates one of the programs and *file_name* the name of the file in which it is wished to save.

4.4 Summary

The chapter presented the prototype tool implementing the approach proposed in this thesis. The Prolog language has been used for the implementation because the characteristic of

versatility and ease of expansion. The existing tools, implementing the ADT method, has been illustrated. This tool has been extended to produce all the facts expressing the structure of the modules identified in an existing system to be reusable. The extension has been presented with the complete set of programs of which it is composed and all the facts that it produces.

In Appendix B all the database produced by the execution of the prototype tool with the program Editor.pas, one of the case studies that will be presented in the next section.

```
2 ?- pm_mod_def_proc_dependency(true).
needs(pm_mod_def_proc, mod_proc_comp)
needs(pm_algo_mod_proc_comp, mod_proc_comp)
needs(pm_algo_mod_proc_comp, dom_proc)
needs_call(pm_proc_pn, ident_useful_proc_comp, pm_algo_mod_proc_comp)
needs(pm_mod_def_proc, proc_pn)
needs(pm_algo_mod_proc_comp, mod_proc_comp)
needs(pm_algo_mod_proc_comp, dom_proc)
needs_call(pm_proc_pn, ident_useful_proc_comp, pm_algo_mod_proc_comp)
needs(pm_dir_dom_comp, proc_pn)
needs(pm_dir_dom_comp, dom_proc)
needs(pm_mod_def_proc, dir_dom_comp)
needs(pm_algo_mod_proc_comp, mod_proc_comp)
needs(pm_algo_mod_proc_comp, dom_proc)
needs_call(pm_proc_pn, ident_useful_proc_comp, pm_algo_mod_proc_comp)
needs(pm_dir_dom_comp, proc_pn)
needs(pm_dir_dom_comp, dom_proc)
needs(pm_mod_call_proc, dir_dom_comp)
needs(pm_algo_mod_proc_comp, mod_proc_comp)
needs(pm_algo_mod_proc_comp, dom_proc)
needs_call(pm_proc_pn, ident_useful_proc_comp, pm_algo_mod_proc_comp)
needs(pm_mod_call_proc, proc_pn)
needs(pm_mod_call_proc, mod_proc_comp)
needs(pm_mod_def_proc, mod_call_proc)

Yes
2 ?- pm_comp.

Yes
3 ?- pm_dom_proc.

Yes
4 ?- pm_algo_mod_proc_comp.
nb_iteration_mod_proc_comp(7)
change(pm_algo_mod_proc_comp, proc_func_call)

Yes
5 ?- pm_mod_def_proc_dependency(true).
needs(pm_mod_def_proc, proc_pn)
needs(pm_dir_dom_comp, proc_pn)
needs(pm_mod_def_proc, dir_dom_comp)
needs(pm_dir_dom_comp, proc_pn)
needs(pm_mod_call_proc, dir_dom_comp)
needs(pm_mod_call_proc, proc_pn)
needs(pm_mod_def_proc, mod_call_proc)

Yes
```

Table 4.2: Unix script for query dependency

Chapter 5

Case Studies

5.1 Introduction

To show the validity of the method proposed in the previous chapters a set of Pascal programs are used in a series of experiments. These programs were analysed by Canfora *et.al.* [23] to test the strength of the ADT method and the results obtained have been used to evaluate, through the values of the quality attributes, the quality of that method. It will be shown how improved results are obtained with the new process. Moreover, it will be shown how the new method facilitates the splitting of a software system, no matter how complex, into more than one module. For each of the new modules obtained, a re-engineered ADT, proposable for reuse, will be provided.

The data for the experiments consist of four Pascal programs developed in different periods and by different people. Clearly, for the different expertise of the developers the programs present different characteristics but the results obtained are comparable.

All the programs analysed have a size between 1000 and 2000 LOC. A brief description and the relative analysis for each program follows. For each program the structure of the identified modules are presented. The notation adopted here uses EXPORT to indicate the *interface specification*, that is all the information, the names of the types involved and the operations that act on those types, that the module exports; and uses BODY for the *implementation*, that is the representation of the type, the local resources and the implementation of the operations on the types, that is, all the information that the module hides.

The final section will show the study carried out for the evaluation of the quality of the new method. The four quality attributes, ***adequacy***, ***method completeness***, ***purity*** and ***module completeness***, will be calculated on the basis of the results obtained from the experiments. The values that will be obtained will show the improvements achieved with the introduction of the extended method.

5.2 Editor.pas

This is a version of the Unix text editor from the Software Tools book [38]. The program is augmented with functions that have been written for a particular environment to carry out operations such as opening files and detecting interrupts from the user. The size of the program is just under 2000 LOC .

The application of the first step of the algorithm, that is the application of the method for the extraction of the candidate modules criterion gave the following results:

```
Module 1
STRUCTURE:   tracestring
OPERATIONS:  etrace
Module 2
STRUCTURE:   lineptr
OPERATIONS:  alloline,freeline,getind,getnew,getpak,gettxt,linkup

STRUCTURE:   argstring,filenamestring,linestring,patternstring,statusrange
OPERATIONS:  addset,amatch,append,assignfile,catsub,ckp,ctoi,defalt,delete,
             dolist,doprnt,doread,dowrit,dumppat,esc,filset,getccl,getfn,
             getnum,getone,getrhs,inject,injpak,locate,makpat,maksub,match,
             move,omatch,open,patsiz,ptscan,readcmd,readline,readterm,
             stclos,subst
```

The first two modules obtained are well-formed; in fact, it is possible to assign a meaning to them: the first module *TraceString* appears very simple with a debugging routine as the only operator; the second module implements the type *Lines* as a pointer to a list of lines. Each node in the list contains information about a text line, that is the content of the line, its length, a field to indicate its logical deletion and two pointers to the next and the previous

lines. The module contains the operations to manage two lists of lines, the list of the used lines (the ones that currently contain text) and the list of the free lines.

The last module appears as a large pot pourri module consisting of five user-defined data types and thirty seven procedure-like components. The analysis of the code revealed that the main cause of this clusterization was the use of the user-defined data type *statusRange* that is shared by numerous procedure-like components. This type codifies the states at the end of a number of different operations. Since *statusRange* is an enumeration type and, by proceeding according to **Step 1** above, it is possible to isolate it and to re-apply the criterion without taking into account this variable.

Carrying out this step leads to the splitting of this module in the following way:

```
Module 3
STRUCTURE:   filenamestring
OPERATIONS:  assignfile,doread,dowrit,getfn,open
Module 4
STRUCTURE:   linestring
OPERATIONS:  addset,ctoi,esc,filset,inject,readcmd,readline,readterm
Module 5
STRUCTURE:   argstring,patternstring
OPERATIONS:  amatch,catsub,dumppat,getccl,getrhs,locate,makpat,maksub,match,
             omatch,patsiz,stclos,subst
```

The candidate ADTs thus obtained, implemented as packed arrays, are: *Files* with all the primitives to manage a file; *LineStrings*, with the operators for reading a line from either the terminal or a file buffer, modifying a piece of text and inserting an escape character; and *PatternMatcher* with the operators for searching and substituting strings, and of pattern matching.

Figure 5.1 shows the CDG of the program Editor.pas. This graph does not contain strongly connected subgraphs.

Here the CDAG coincides with the CDG.

Figure 5.2 shows the SDDT of the program.

In the figure, the solid lines represent relation of strong direct dominance, and the dashed

Figure 5.1: Call Graph of Editor.pas

Figure 5.2: Strongly Direct Dominance Tree of Editor.pas

lines the relation of direct, but not strong, dominance. According to **Step 3** of the informal algorithm, all the nodes representing components belonging to some module are labelled.

On the contrary, the nodes not labelled are those not belonging to any module. The labels of the modules are progressively numbered corresponding to the order of their presentation above (TraceString is labelled 1, Lines is labelled 2, and so on).

By observing the SDDT in Figure 5.2, it can be noticed that three procedure-like components (*doglob*, *ckglob* and *initialize*), not belonging to some module, are strongly dominated from the MAIN program. According to **Step 4** the nodes corresponding to these components can be deleted. The obtained CDAG is not showed, but the new SDDT is showed in the Figure 5.3.

Figure 5.3: SDDT of Editor.pas after first iteration

In this new SDDT the procedure-like components *prevln* and *docmd* are strongly dominated from the MAIN program. Reiterating **Step 4**, the nodes representing them can be deleted from the previous CDAG. In a next reiteration, the nodes representing the components *defalt*, *ckp*, *move*, *injpak*, *dolist* and *doprnt* will be deleted because they are strongly dominated, and, then, called only from *docmd* and, because they do not belong to some candidate module. As indicated in **Step 5**, the above deletion brings to the creation of assumed arcs between the node MAIN and the nodes that, in the new situation, have no incoming arcs, because they previously connected only from deleted nodes. Examples of these nodes are *gethrs*, *getone*, *getfn*, *dowrit*, and so on.

Figure 5.4 and Figure 5.5 show the SDDTs obtained respectively after the above process and after a new iteration of **Steps 4**, **5** and **6**.

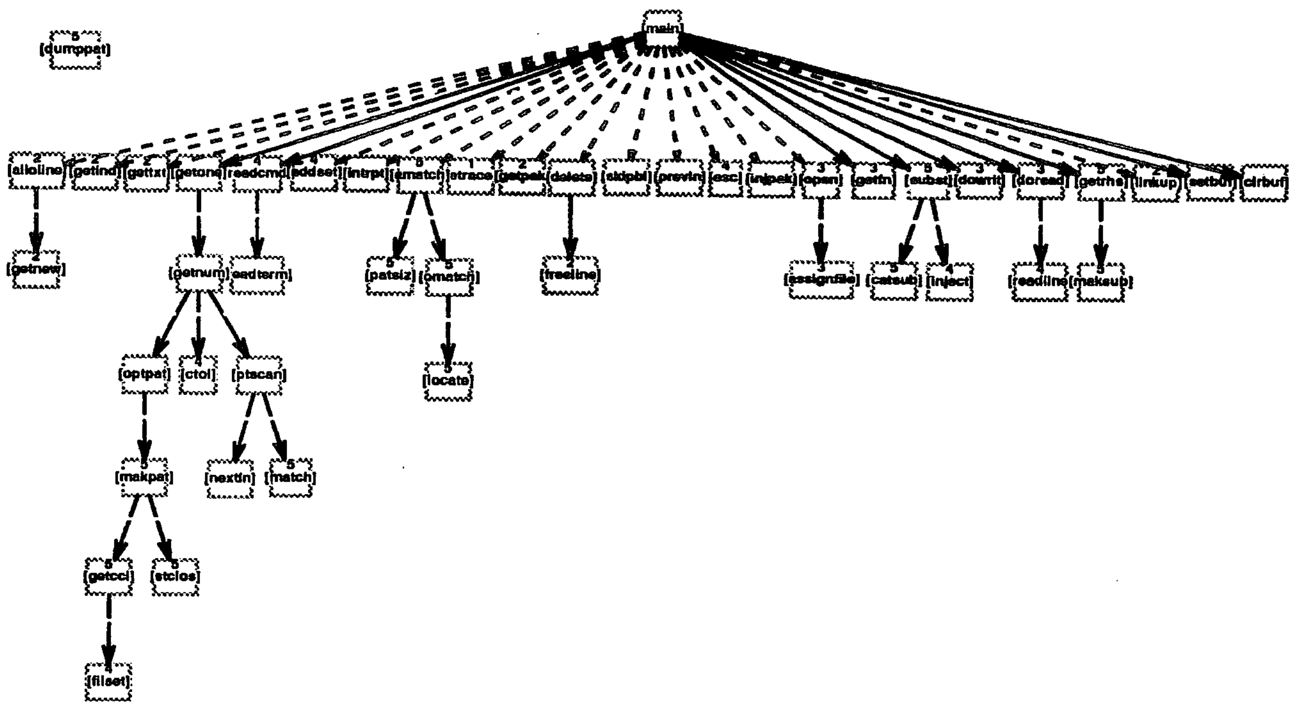

Figure 5.4: SDDT of Editor.pas after third iteration

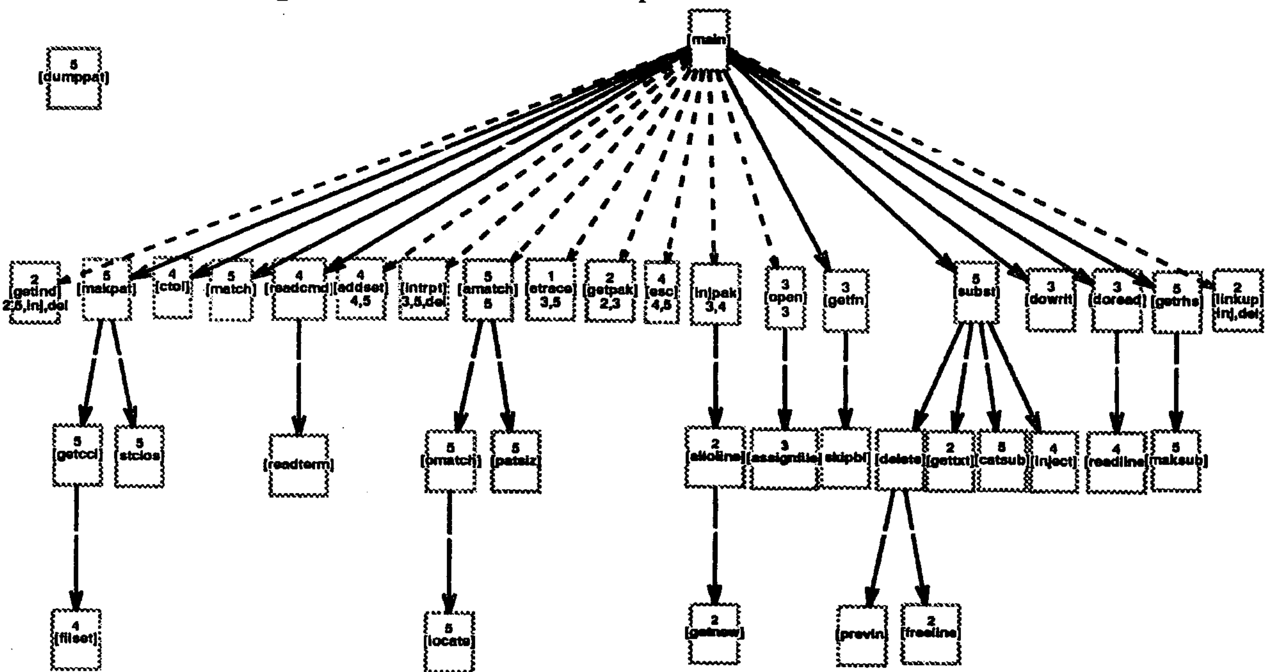

Figure 5.5: SDDT of Editor.pas after last iteration

In particular, some of the nodes in the last SDDT bring additional labels. In fact, following **Step 7**, all the procedure-like components represented from nodes that are not strongly

dominated are called from more than one procedure-like component, and for each of these nodes the additional label indicates either the index of the modules which contain the calling components, or the name of the calling components if it does not belong to some module.

In Figure 5.5, the final SDDT obtained shows all the relations of call existing between the procedure-like components belonging to the modules and between these and other components not belonging to any module. For example, it appears that the component *esc* belongs to the 4th module, that is *LineString*, and it is called from components of the 4th module and the 5th module, that is *PatternMatcher*. On the basis of **Rule 5**, the last assertion settles that the 5th module will USE the 4th. Analogously, the procedure-like component *injpak* does not belong to some module but it is called from components of the 3rd (*Files*) and the 4th modules, then, from **Rule 6**, its declaration can be placed in both of the modules. In particular *injpak* is called only from one component inside each module and, for this reason the component will be nested in the calling components.

To complete all the process, the next step is to analyse the final SDDT. Separate considerations for each module will be given.

The SDDT shows that the only component in ***module 1***, ***TraceString***, does not call any other components, but it is called from components of the 3rd and 5th modules. So, from **Rule 5**, there will be a relation of USE from the 3rd and 5th modules to the 1st. This module appears quite simple:

```
module MOD1
---------------------------------------------------------  EXPORT
type   traceString;

       procedure etrace(traceName : traceString);
---------------------------------------------------------  BODY
type   traceString = Packed array [1..6] of char;

       procedure etrace(traceName : traceString);
            begin ... end;
---------------------------------------------------------
```

The procedure-like components of ***module 2*** (***lineptr***), do not call other procedure-like components with the exception of components of the same module. Thus, the 2nd module does not use any other modules. The subtree formed from ***alloline*** and *Getnew* is a MET subtree and, according to **Rule 1**, the second component can be nested in the first. But, careful analysis of the code reveals that it is important to export both of the components. Moreover, the component *getind* is called from ***getpak*** and this from ***gettext***. This fact will justify the order of their declaration (**Rule 4**). The module obtained is the following:

```
module MOD3
USES MOD1,MOD2,MOD4
type    fileNameString;statusRange;                                                    EXPORT
        function open(fileName:fileNameString;dir:integer):integer;
        function dowrit(fromLine,toLine:integer;fileName:fileNameString):statusRange;
        function doread(line:integer;fileName:fileNameString):statusRange;
        function getfn(var fileName : fileNameString) : statusRange;
                                                                                       BODY
const   .................
type    fileNameString = Packed array [1..MAXNAME] of char;
        statusRange = (NOSTATUS, OKSTATUS, ERRSTATUS, EOFSTATUS, INTSTATUS);
        function open(fileName:fileNameString;dir:integer):integer;
           function assignFile(var FILEX:text;var strname:fileNameString;
                    dir:integer):integer;
        function intrpt(var x : integer) : boolean;
        function dowrit(fromLine,toLine:integer;fileName:fileNameString):statusRange;
        function doread(line:integer;fileName:fileNameString):statusRange;
           function injpak(var sts : statusRange) : statusRange;
        function getfn(var fileName : fileNameString) : statusRange;
           procedure skipbl;
```

Considerations analogous to those above give the structure for *module 4*. An additional consideration is that *module 4* USES *module 2*, because the component *injpak*, called from *module 4* and calling two components of *module 2*, will belong *module 4* from **Rule 6**. *Module 4* is:

```
module MOD4
USES MOD2
type    lineString;                                                                    EXPORT
        function addset(c:char; var str:lineString; var j:integer;
                 maxsiz:integer):boolean;
        function ctoi(lin:lineString; var i:integer):integer;
        function esc(str:lineString; var i:integer):char;
        procedure readline(var FILEX:text; var lin:lineString; var lineLen:integer);
        procedure filset(delim:char; lin:lineString; var i:integer; var str:lineString;
                 var j:integer; maxstr:integer);
        function readcmd(var lin:lineString; promptCh:char):boolean;
        function inject(lin : lineString) : integer;
                                                                                       BODY
const   .................
type    lineString = Packed array [1..MAXLINE] of char;
        function addset(c:char; var str:lineString; var j:integer;
                 maxsiz:integer):boolean;
        function ctoi(lin:lineString; var i:integer):integer;
        function esc(str:lineString; var i:integer):char;
        procedure readline(var FILEX:text; var lin:lineString; var lineLen:integer);
        procedure filset(delim:char; lin:lineString; var i:integer; var str:lineString;
                 var j:integer; maxstr:integer);
        function readcmd(var lin:lineString; promptCh:char):boolean;
           type
              promptString = Packed array [1..MAXPROMPT] of char;
              promptRec = record
                 len : integer;
                 str : promptString;
              end;
           function readterm(var lin:lineString; var len:integer;
                 var prompt:promptRec; var readlen:integer):integer;
        function inject(lin : lineString) : integer;
type    statusRange = (NOSTATUS, OKSTATUS, ERRSTATUS, EOFSTATUS, INTSTATUS);
           function injpak(var sts : statusRange) : statusRange;
```

Finally *module 5* appears more complex: it USES *modules 1, 2* and *4*, includes components like *delete* and *prevl* that, in the structure of the module, will be nested in *subst*, and it

contains more complex functionalities composed of sub-functionalities belonging to the same module like *amatch* that implements the sub-functionalities implemented by *omatch*, *patsiz* and *locate* that can be nested in the dominant component. By continuing to apply the rules as above the structure of this ADT becomes:

```
module MOD5
USES MOD1,MOD2,MOD4
                                                                              EXPORT
type
      argString; patternString;
      function amatch(lin:argString; from:integer; var pat:patternString):integer;
      function makpat(arg:argString; from:integer; delim:char;
            var pat:patternString):integer;
      function match(lin:argString; var pat:patternString):boolean;
      function subst(sub:patternString; gflag:boolean):integer;
      function getrhs(var sub:patternString; var gflag:boolean):integer;
      procedure dumppat(pat : patternString; j : integer);
                                                                              BODY
const ............
type
      argString = lineString;
      patternString = lineString;

      function amatch(lin:argString; from:integer; var pat:patternString):integer;
         function omatch(lin:lineString; var i:integer; pat:patternString;
               j : integer) : boolean;
           function locate(c:char; pat:patternString; offset:integer):boolean;
         function patsiz(pat:patternString; n:integer):integer;
      function makpat(arg:argString; from:integer; delim:char;
            var pat:patternString):integer;
         function getccl(arg:argString; var i:integer; var pat patternString;
               var j:integer):boolean;
         function stclos(var pat:patternString; var j,lastj,lastcl:integer):integer;
      function match(lin:argString; var pat:patternString):boolean;
      function subst(sub:patternString; gflag:boolean):integer;
         type statusRange = (NOSTATUS, OKSTATUS, ERRSTATUS, EOFSTATUS, INTSTATUS);
         function delete(fromLine,toLine:integer):statusRange;
           function prevln(line:integer):integer;
         procedure catsub(lin:lineString; f,t:integer; sub:patternString;
               var str:lineString; var k:integer; maxnew:integer);
      function getrhs(var sub:patternString; var gflag:boolean):integer;
        function maksub(arg:argString; f:integer; delim:char; var sub:patternString)
               :integer;
      procedure dumppat(pat : patternString; j : integer);
```

5.3 ExamMarker.pas

The program implements a system for the evaluation of multiple choice examinations. It is particularly suitable for Universities that adopt a college organization like that of the University of Durham. The program inputs the number of questions, with the alternative answers and the exact answers, and the information about the students, with their answers, and produces the resulting marks in various orders.

The first application of the ADT method, bring to the identification of the following six

modules:

```
Module 1
STRUCTURE:   strings
OPERATIONS:  readstring
Module 2
STRUCTURE:   colleges
OPERATIONS:  newcollege,readcollege,writecollege
Module 3
STRUCTURE:   listelements
OPERATIONS:  alphaprecede,collegeprecede,highermark,swap
Module 4
STRUCTURE:   candidates,lines,papers
OPERATIONS:  checkavailablealternatives,checkcandidate,checkextradata,
             readandcheckanswers,writeparticulars
Module 5
STRUCTURE:   listsizes,markfudge,markschemes,questnos,titles
OPERATIONS:  getpreliminaries,readtitle,writetitle

STRUCTURE:   exams,lists,posint,seeds
OPERATIONS:  analyse,dice,dumptofile,getparticulars,getrandomnumber,histogram,
             listbycollege,listbymark,listbyname,listforstudents,
             listsortedresults,mark,permute,quicksort,summarise,validate
```

It is possible to recognize a meaning in the first five modules. *Strings* implements the type string as a packed array of char. It has the only operator ReadString to read a string to a maximun length from a file. *Colleges* implements the type college as an enumeration type and it has the operators to read and write the college and the code in a source file. *ListElements* clusters the type ListElements with the operators to compare students records on the basis of alphabetical order of their name, their marks or the college which they belong. *CandidatePapers* contains the operators for reading and checking the data of the students and their answers and printing out the resulting mark. *Script* implements the type *Title* with the operators to read and to write a title, and contains also the operators to read all the information related to the exam like the title of an exam, the total number of questions, the marking scheme, and to produce the input echo-printed into a file.

Also in this case, a quite large module is obtained. The analisys of its code does not permit the association of this module to an ADT. The existence of two subrange types, *seeds* and *posint*, suggests the cause of the clusterization of more one modules in a unique one. Those two types are used to randomly rearrange the members of a class when writing into a file the marks obtained for each question in the examination, thus enforcing the anonymity of data, and they can be declared locally in the unique procedure-like component that refer them, *DumpToFile*. The isolation of those two types and the re-application of the ADT method split the last module in the following:

```
Module 6
STRUCTURE:   lists
OPERATIONS:  histogram,listbycollege,listbymark,listbyname,listforstudents,
             permute,quicksort
Module 7
STRUCTURE:   exams
OPERATIONS:  analyse,dumptofile,getparticulars,listsortedresults,mark,
             summarise,validate
```

The two modules above implement, respectively, the ADT *Lists*, and the ADT *Exams*. The structure of the first is implemented as a record containing the list of all the candidates and the respective answer to the tests, and it contains all the operators to list the results of an exam in different orders on the basis of the college, the mark, the name, and the students. To do all that it includes the operators *quicksort* to order in different ways and *permute* to exchange two elements. The operator to draw the histogram of the results, selected by the method, will not be considered here because the program does not contain the implementation of the functionality. The second ADT is composed of a structure implemented as a record containing all the information necessary for an exam: the title, the questions with the correct answers, the list of the candidate and so on, and all the operators to manage an exams.

Figure 5.6 presents the CDG and CDAG of the program Exammaker.pas, while Figure 5.7 shows the final SDDT.

Only the final SDDT is shown. In the operation of deletion of components not belonging to modules, only one component, *initialise*, has been deleted. The procedure *initialise* is

Figure 5.6: Call Graph of ExamMaker.pas

Figure 5.7: Final Strongly Direct Dominance Tree of ExamMaker.pas

a component, called from MAIN, executing the necessary initialisation at the start of the program.

Module 1 present a quite easy structure, and as it can be observed in the final SDDT it does not use any other module, and it is used only from the *module 4*.

Module 2, implementing the type *Colleges*, as well as the components already discovered includes also the components *lower* and *capital* with the first nested in the operator *readcollege* because strongly direct dominated from it. The structure of *module 2* is the following:

```
module MOD2
                                                                                EXPORT
type  colleges;

      PROCEDURE ReadCollege (VAR college:colleges;VAR unknown:boolean;VAR source,echofile: text);
      PROCEDURE WriteCollege (college:colleges;VAR outfile:text);
      PROCEDURE NewCollege (college:colleges;VAR lastcollege:colleges;VAR outfile:text);
                                                                                BODY
type  colleges = (grey,collingwood,marys,trevs,mildert,aidans,hatfield,
             chads,johns,cuths,castle,hildandbede);

      FUNCTION capital (ch: char): char;
      PROCEDURE ReadCollege (VAR college:colleges;VAR unknown:boolean;VAR source,echofile: text);
        FUNCTION lower (ch:char):char;
      PROCEDURE WriteCollege (college:colleges;VAR outfile:text);
      PROCEDURE NewCollege (college:colleges;VAR lastcollege:colleges;VAR outfile:text);
```

Module 3, implementing the type *ListElements*, gives an easy structure. It defines and exports all the procedure-like components surveyed by the ADt method, in fact each of them is used from other modules. This module does not use any other module but it is used from *module 6* and *module 7*.

As it is revealed in Figure 5.7 from the number 4 under the name of the component *writecollege*, *module 4* uses *module 2*. Besides, it includes in its implementation the declaration of the components *seekchar* and *checkterminator*, whose declaration will be added also in *module 7*; in fact, the ADT method does not include these procedure-like components in any module but they are used from procedure-like components of *module 4*, wilst the definition of *checkterminator* will be included also in *module 7*. The structure of this module is the following:

```
module MOD4
                                                                                          EXPORT
type    candidates,lines,papers;

        PROCEDURE CheckAvailableAlternatives (VAR paper:papers;VAR errline:lines;VAR valid:boolean;
                  VAR source,echofile:text);
        PROCEDURE CheckCandidate (VAR candidate:candidates; paper:papers;VAR valid:boolean;
                   VAR echofile: text);
        PROCEDURE CheckExtraData (VAR errline:lines;VAR present:boolean;VAR source,echofile:text);
        PROCEDURE PROCEDURE ReadAndCheckAnswers (VAR script:scripts;paper:papers;VAR valid:boolean;
                  VAR source, echofile: text);
        PROCEDURE WriteParticulars (candidate:candidates;collegerequired,markrequired:boolean;
            percentrequired,attemptsrequired:boolean;VAR outfile:text);
                                                                                          BODY
type    lines = PACKED ARRAY [linepos] OF char;
        papers = RECORD
        ...............
        END {papers};
        candidates = RECORD
        ..............
        END {candidates};

        PROCEDURE SeekChar (VAR requiredchar:char;firstchar,lastchar: char;
             VAR inrange:boolean;VAR source,echofile:text);
        PROCEDURE CheckAvailableAlternatives (VAR paper:papers;VAR errline:lines;VAR valid:boolean;
                  VAR source,echofile:text);
        PROCEDURE CheckCandidate (VAR candidate:candidates; paper:papers;VAR valid:boolean;
                   VAR echofile: text);
        PROCEDURE PROCEDURE ReadAndCheckAnswers (VAR script:scripts;paper:papers;VAR valid:boolean;
                  VAR source, echofile: text);
          PROCEDURE CheckTerminator (VAR source,echofile:text;terminator:char;
                    VAR correct:boolean);
        PROCEDURE CheckExtraData (VAR errline:lines;VAR present:boolean;VAR source,echofile:text);
        PROCEDURE WriteParticulars (candidate:candidates;collegerequired,markrequired:boolean;
            percentrequired,attemptsrequired:boolean;VAR outfile:text);
```

Module 5 does not use other modules but it is used form *module 7*. It does not include other procedure-like components different from those selected by the ADT method.

Module 6, implementing the type *Lists*, includes the declaration of the two subrange types isolated to split the last module obtained by the first application of the ADT method and the procedure-like components, *dice*, and *getrandomnumber*, that have been discarded in consequence of that isolation; they are nested in *permute*. For its implementation the module uses *modules 2* and *3*.

```
module MOD6
USES MOD2,MOD3
                                                                              EXPORT
type    lists;

        PROCEDURE ListByName(class:lists;VAR outfile:text);
        PROCEDURE ListForStudents (class:lists;VAR outfile: text);
        PROCEDURE ListByCollege (class:lists;VAR outfile:text);
        PROCEDURE ListByMark (class:lists;maxposs,minposs:marks;VAR outfile:text);
        PROCEDURE Permute (VAR list:lists; n:listpos);
        PROCEDURE QuickSort (VAR list:lists;leftend,rightend:listpos;
           FUNCTION precede (element1,element2:listelements):boolean);
                                                                              BODY
const   .......
type    listpos = 1..maxcand;
m       listsizes = 0..maxcand;
        listelements = ĉandidates;
        lists = RECORD
           contents: ARRAY [listpos] OF listelements;
               size:listsizes;
           END lists;

        PROCEDURE ListByName(class:lists;VAR outfile:text);
        PROCEDURE ListForStudents (class:lists;VAR outfile: text);
        PROCEDURE ListByCollege (class:lists;VAR outfile:text);
        PROCEDURE ListByMark (class:lists;maxposs,minposs:marks;VAR outfile:text);
        PROCEDURE Permute (VAR list:lists; n:listpos);
           PROCEDURE Dice (n:posint;VAR result:integer;VAR seed:seeds);
             const ...........
             type posint = 1..maxint;
               seeds = 0..maxseed;
             PROCEDURE GetRandomNumber (VAR random: real; VAR seed: seeds);
        PROCEDURE QuickSort (VAR list:lists;leftend,rightend:listpos;
           FUNCTION precede (element1,element2:listelements):boolean);
```

The final module, 7, is the fundamental module. By using all the other modules, with the exception of *module 1*, it implements the type *exams*. It manages the whole exam, with operator to use *module 5* to input the information about the exam, *module 2* to input information about the colleges, *module 4* to input information about the candidates and their answers, *module 3* to verify the validity of the information acquired, *module 5* to list the candidates and their answers in different way, and the operator to analyse the exam, assign values and validate the legality of all the terms.

5.4 Minicalc.pas

This program implements a simple spreadsheet and has been taken from a text book [40]. It is provided with a simple portable interactive user interface, and shows the display divided into cells, labelled A to H vertically and 1 to 5 horizontally. The system takes as input the user commands that can be either commands for the management of the spreadsheet or information to input into the cells.

The analysis of this program [23], gave interesting results, but they were not complete in

terms of operations presented for each module. In fact, the ADT method was applied to only the procedure-like components declared in MAIN, and it did not take into consideration operations that are fundamental for some ADTs but are implemented by components nested in others. Different results have been obtained by not considering the nesting between components in the first stage of the analysis. When compared with the previous results, the new results appear richer in terms of operations for each module.

The first results obtained are the following:

```
Module 1
STRUCTURE:   usermsgs
OPERATIONS:  writeuser,errorhandler
Module 2
STRUCTURE:   commands
OPERATIONS:  getcommand

STRUCTURE:   cellid,colindex,counter,inputtype,nodeptr,rowindex,token
OPERATIONS:  addtodependlist,alphabetic,checkexprtree,docellchange,
             docellexpr,docelllabel,evaluate,expression,factor,getcell,
             getchar,getexpression,getinp,gettoken,getuserinput,makenode,
             movetocell,numeric,parseexpression,postlabel,postvalue,skipblanks,
             subexpr,term,ungetchar,writeexpression,writeuserinput
```

The first two modules are very simple, but the last clusters together software components not in a really definite way and is a huge module. The isolation of the subrange types and the re-application of the ADT method brought simpler modules, but equally complex, composed of the set of all the components that appear in the last three modules. Human intervention has been necessary to reach better results and the splitting of the module can be obtained by analysing the *call* relations between the procedure-like components and those existing between them and the user-defined data types. The clusterization is due to components, *makenode*, *checkexprtree* and *parseexpression* that implement the characteristic functionalities to manage and to evaluate algebraic expressions and are strongly dominated from other components that have the same purpose, but use in the interface the types *inputtype,token* and *cellid*, that are referred from other procedure-like components implementing a different kind of functionality. The deletion of the links existing between the last procedure-like components and those implementing functionalities in algebraic expressions led to the splitting

of the module into the following three modules:

```
Module 3
STRUCTURE:   inputtype,token
OPERATIONS:  writeuserinput,getcell,gettoken,alphabetic,skipblank
Module 4
STRUCTURE:   nodeptr
OPERATIONS:  checkexprtree,evaluate,expression,factor,getexpression,
             makenode,parseexpression,subexpr,term,writeexpression
Module 5
STRUCTURE:   cellid
OPERATIONS:  docellexpr,docelllabel,docellchange
```

Figure 5.8 shows the CDG of Minicalc.pas.

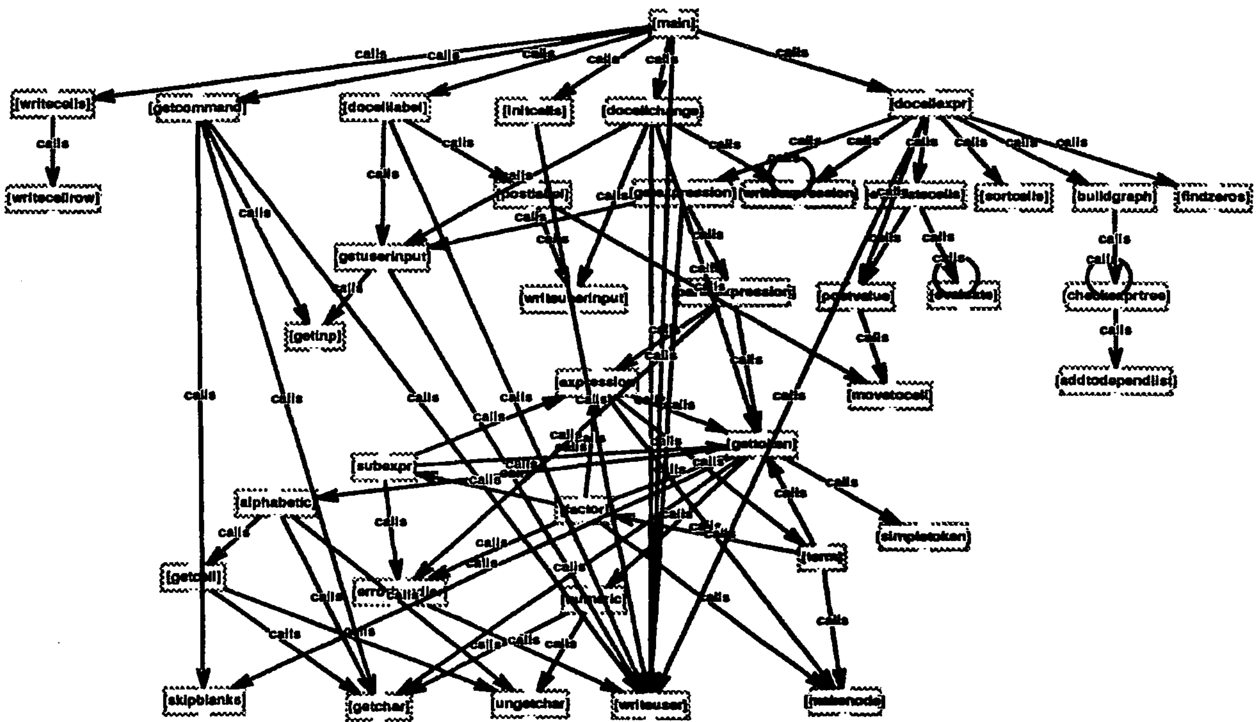

Figure 5.8: Call Directed Graph of Minicalc.pas

By observing the CDG it is possible to identify the strongly connected subgraph formed by the procedure-like components *expression*, *subexpr*, *term* and *factor*. All these procedure-like components belong to *module 4* implenting a functionality of this module. Figure 5.9 shows the CDAG. Here, the node EXPRESSION substitutes the strongly connected subgraph; its

Figure 5.9: Call Directed Acyclic Graph of Minicalc.pas

label indicates that the collapsed subgraph implements a functionality of the ADT *Expression* that will be later analysed.

Finally, Figure 5.10 shows the final SDDT obtained after the deletion of uninteresting components. From the first SDDT, only three components used to initialise the system, were discarded.

As can be seen the components in *module 1* do not call other components but are used from other modules. *Module 1* is very simple, it implements the type *User_Messages* managing the messages to the user.

The second module, *User_Commands*, contains the operator to read and to interpret the user commands, and uses *module 1* and operations of *module 3*.

Module 3 implements the type *Input* with the operators to manage all the different kinds of inputs, that is commands, or information to store in the cells of the spreadsheet. This information can be a label, or numeric values, or references to other cells. The module contains operations to verify the correctness of the information by analysing eventual links between cells. The final SDDT reveals that some components (*alphabetic*, *getcell*, *numeric*, etc.) implement sub-functionalities of another component (*gettoken*), and that they can be nested in the latter. Also, the component *simpletoken* can be nested here since it is strongly

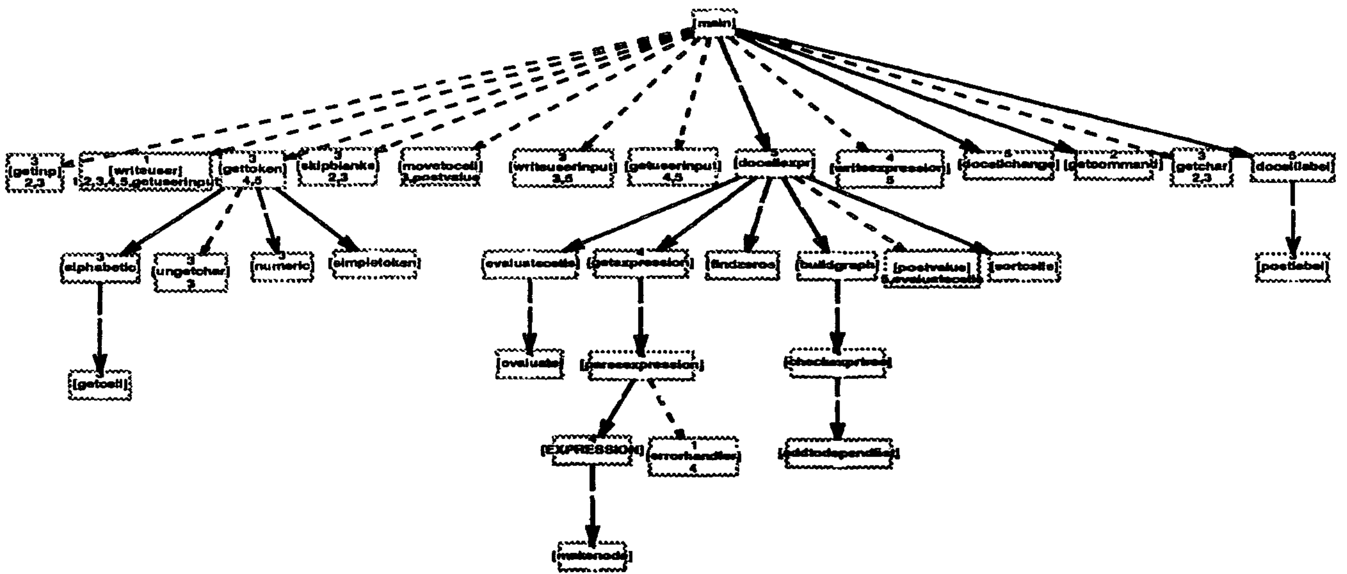

Figure 5.10: Final Strongly Direct Dominance Tree of Minicalc.pas

dominated from *gettoken.* This was not apparent from the first analysis and was thus not added to the module, The routine *movetocell* called from *postlabel* will also be inserted in this module. It appears that the subject module uses *module 1.* These and other observations that can spring from Figure 5.10 give the final structure of the module as:

```
module MOD3
type    InputType,Token;                                                          EXPORT
        function GetInp(var UserInp:InputType):boolean;
        function SkipBlanks(var UserInp:InputType):boolean;
        procedure GetChar(var UserInp:InputType;var Ch:char);
        procedure GetToken(var UserInp:InputType;var NextToken:Token;
                          var Num:real;var Field:CellId);
        procedure WriteUserInput(UserInp:InputType);
        function GetUserInput(var UserInp:InputType):boolean;
        procedure PostLabel(CellRow:Integer;CellCol:integer;AValue:InputType);
                                                                                  BODY
const   .................
type    Token=(SinFunc,CosFunc,ExpFunc,LnFunc,SqrtFunc,UMinus,Plus,Minus,
            Times,Divide,CellLabel,LeftParen,RightParen,Number,
            NoMore,Unknown,ACellName,BadCell);
        InputType = record
            Line : LineType;
            Length,last : Counter;
        end;
        ...................;
        function GetInp(var UserInp:InputType):boolean;
        function SkipBlanks(var UserInp:InputType):boolean;
        procedure GetChar(var UserInp:InputType;var Ch:char);
        procedure GetToken(var UserInp:InputType;var NextToken:Token;
                          var Num:real;var Field:CellId);
           procedure UnGetChar(var UserInp:InputType;var Ch:char);
           function Alphabetic (var Ch:char;var UserInp:InputType; var Field:CellId):Token ;
              function GetCell(var Ch:char;var UserInp:InputType;var Field:CellId):boolean;
           function Numeric(var Ch:char;var UserInp:InputType):real;
           procedure SimpleToken(Ch:char);
        procedure WriteUserInput(UserInp:InputType);
        function GetUserInput(var UserInp:InputType):boolean;
        procedure PostLabel(CellRow:integer;CellCol:integer;AValue:InputType);
           procedure MoveToCell(CellRow:integer;CellCol:integer);
```

Module 4 is interesting in that it implements the ADT *Expression* with the operators to read into the spreadsheet, to write, to check the correctness of and to evaluate an expression. In particular, the operation to read an expression is implemented by the components *expression*, *subexpr*, *term* and *factor* that constitute the strongly connected subgraph in the CDG collapsed in the node called *EXPRESSION*. It will be nested in *parseexpression* that will be nested in *getexpression*. The module will include also the component *addtodependlist* nested in *checkexprtree*. *Module 4* uses *module 1* and *module 3*. In the module the type *counter* is also defined; this type is used from some components to consider the depth in the representation of the managed expression like binary tree. The above information is synthesized in the following schema:

```
module MOD4
USES MOD1, MOD3
type    nodeptr;counter;                                                    EXPORT

        procedure WriteExpression(Expr:NodePtr;Level:Counter);
        function GetExpression:NodePtr;
        function Evaluate(Expr:NodePtr;var Defined:boolean):real;
        function CheckExprTree(ExprTree:NodePtr;var Count:Counter):boolean;
                                                                            BODY
const   .................
type    Counter = 0..maxint;
        LineType = array[1..MAXLINE]of char;
        InputType = record
              Line : LineType;
              Length : Counter;
              Last : Counter;
        end;
        procedure WriteExpression(Expr:NodePtr;Level:Counter);
        function GetExpression:NodePtr;
           function ParseExpression(UserInp:InputType):NodePtr;
              function MakeNode(NodeValue:real;Left, Right:NodePtr):NodePtr;
              function Expression:NodePtr;
                 function Term : NodePtr;
                    function Factor : NodePtr;
                       function SubExpr : NodePtr;
        function Evaluate(Expr:NodePtr;var Defined:boolean):real;
        function CheckExprTree(ExprTree:NodePtr;var Count:Counter) : boolean;
           procedure AddToDependList(var Count : Counter);
```

Finally, the last module implements the ADT *Cells*. This module, by using all the other modules, manages the spreadsheet with all the operators for the acquisition and the checking of labels, cell addresses and numeric expressions. In particular, for the last task, the component ***docellexpr*** declares the components ***evaluatecells***, ***buildgraph***, ***sortcells*** and ***findzeroes***, whose names express the kinds of implemented operations. Also in this case, a real partition of the software components in more than one hierarchic module is obtained.

5.5 Format.pas

This is an ancient public domain pretty printer program for Pascal that has been changed and added to by a number of different people.

The previous application of the method based on ADTs to the Format program produced very poor results. Three modules were obtained with a very simple structure.

This time, the first incomplete modules obtained are the following:

```
Module 1
STRUCTURE:   alpha,symbols
OPERATIONS:  checkfor,dostmtlist,insertsymbol
Module 2
STRUCTURE:   commenttext,width
OPERATIONS:  doblock,dostatment
Module 3
STRUCTURE:   margins
OPERATIONS:  changemarginto
Module 4
STRUCTURE:   symbolset
OPERATIONS:  dodeclarationuntil, dofieldlistuntil
Module 5
STRUCTURE:   optionsize
OPERATIONS:  bunch
Module 6
STRUCTURE:   params
OPERATIONS:  readin
```

Figure 5.11 shows tha CDG of format.pas. The study of the call graph reveals the existence of the following four strongly connected subgraphes:

```
SCC1:   doblock, doprocedures
SCC2:   dostatement, dostmlist
SCC3:   dorecord, dofieldlistuntil, dovariantrecord
SCC4:   readsymbol, skipcomment, docomment
```

Figure 5.12 shows the SDDT and final SDDT of the Format.pas program.

From the original SDDT only the initialization procedure-like components have been discarded; all the other components cooperate together to the implementation of the functionalities expressed in the identified modules. The six modules obtained look very simple but, in reality, are very complex. It appears that they cannot be re-engineered to be reusable

Figure 5.11: Call Directed Graph of Format.pas

Figure 5.12: Final Strongly Direct Dominance Tree of Format.pas

ADTs because they are lacking in operations, and, moreover, the large use of global variables would make this task very difficult. However, they can offer an useful trace of the complexity of the code. They can be considered as *main* functionalities which are composed of the set of functionalities offered from the whole program. By using each other these modules cooperate in obtaining the goal of the program.

The existence of the strongly connected subgraphs is the main cause of the recursion between the modules. In fact, *SCC2* is composed of a procedure-like component belonging to *module 1* and another one belonging to *module 2*. Since there are two components that are mutually recursive it should be expected that the two modules to be mutually recursive. On the other hand, *SCC4* does not contain components belonging to some module but is used by *module 1,2* and *4*, and uses some of this module, then its components cause the mutual recursion between *module 1,2* and *4*.

The other strongly connected subgraphs are *SCC1* and *SCC3*. The former, having a procedure-like component, *doblock*, belonging to *module 2* and the other one, *doprocedures*, to no module will implement a sub-functionality of *module 2* that will include both the procedure-like components. The subgraph *SCC3*, will belong to *module 4* because it contains one procedure-like component, *dolistfieldunitl*, belonging to this module and the others to no other module. Analogously, the last two components will be included in *module 4* to enrich its structure. Another cross use between modules, and between components, can be seen in the final SDDT. The only modules that are not involved in the recursions and not used from other modules are *module 3, 5* and *6*. The names of the procedure-like components belonging to these modules are eloquent enough to indicate the kind of functionalities implemented. For example, *module 3* is responsible for implementing one of the layout parameterizations, namely the width of the indentation. It is quite difficult to identify the way in which these functionalities are implemented. This is probably caused by the number of authors responsible for writing the program and the unavailability of the original program design.

Format is an important experiment even though poor results were achieved. This is because a partition of the program into simpler modules was obtained, and because it shows the importance of program design and that the loss of documentation is very often the main cause for not understanding a system.

5.6 Evaluation and Conclusion

The new method has been tested on the case studies showed above. The results obtained are more satisfying than those with the experiments on the ADT method, and for which the values for the quality attributes have already been shown. An improvement in the values of the quality attributes has been reached confirming the assertion given above. The values of the quality attributes obtained follow:

Method Completeness	Adequacy	Purity	Module Completeness
90%-100%	90%-100%	90% - 100%	95% - 100%

To demonstrate the validity of these results the case study Editor.pas will be again examined with a comparison with the results for it obtained by the only application of the ADT method. By using the ADT method for that program only two candidate modules were identified; a set of five user-defined data types and thirty six procedure-like components composed a third module, not directly proposed for reuse. On the contrary, with the application of the new method five modules have been identified. The set of procedure-like components not included in any module cannot constitute a new module, then any other module cannot be identified, and the five identified ADTs represent the complete set of ADTs identifiable in the program. This result provides high value in the attribute of method completeness. The equally high value of adequacy is justified from the fact that a semantic has been recognized in all the modules identified. Finally, in the first phase of application of the ADT method and isolation of sub-range and enumeration typoes for each extractable ADT all the operators are recognized, and, in a next step, are localized only and all those procedure-like components used for the implementation of those operators. Each module so obtained contain all the software components necessary for its implementation, that is it represents a complete module, and it does not contain extraneous software components. This justify high value of the attribute of purity.

Analogous considerations can be made for the other case studies.

That some poor results were obtained, like for example for Format.pas, is justified from the fact that choices of project and of structure adopted to construct the original sofware

system, do not always fit with the requests that the method can satisfy. Very often the absence of the program's specification and the design and the lack of knowledge about the way in which the system has been implemented make difficult the analysis of some of the modules obtained and the rejection of them.

The method, presented in the chapter offers also a useful instrument for the comprehension of the code [37]. In fact, it operates, for the large part of the case studies a real division of the software system thus making the subject programs easier to understand. Then, by dividing the subject program into more than one module, each of which implements an abstract data type or a group of functionalities, the method can be used for the comprehension of the code. At the end of the application of the method the program appears as a collection of simpler systems that, for their dimensions, are easier to comprehend than the full program. Also the interactions existing between the obtained modules to the pursuing of the program goal are identified.

Chapter 6

Conclusion

6.1 Introduction

The basic premise of this thesis to analyse the state of the art in the field of the Software Reuse

Software reuse is widely regarded as offering the opportunity for improving the software production process. It is expected that a massive increase in software reuse is the most promising way of overcoming the software crisis. It can lead to substantial increases in productivity and also to software system which are more robust and more reliable. Also if it widespread opinion that the reusable components have to be produced, it is possible to assume that a key method to obtain reusable components consists on extracting and re-engineering them from existing systems. All that entails the definition and setting up of Reverse Engineering and Re-engineering processes. The first kind of processes aims to identify potentially reusable components, while the second kind of processes aim to conform the reusable components to the quality standard, component templates and component interconnection standards established.

Different paradigm for the extraction of reusable components from code have been proposed. This thesis shows the Reuse Re-engineering reference paradigm developed within the RE^2 project. The reference paradigm shows a systematic approach to search the existing software systems for reusable component. Its phases have been presented and, in particular, the candidature phase has been faced. This phase is concerned with the analysis of the source code for the identification of sets of software components that can be candidate to make up

a reusable component. Various methods involved in this phases have been presented, and the analysis of these method has brought to the developement of a more precise method. The formularization, the implementation and the evaluation of the new method have been presented. In the next section a discussion of the criteria for success stated in the introduction is presented and in them it is possible to ensue the validity of the new method.

6.2 Evaluation of the Criteria for Success

Chapter 1 presented a list of the criteria against which this thesis can be evaluated. Each of the criteria are now addressed.

- **Description and evaluation of existing methods for the identification and extraction of reusable abstract data types.**

Several approaches for the identification of reusable assets in code have been proposed in the literature. Each of them looks in the existing code for a particular kind of abstractions in code and the choice of the principle on which they are based is depended on the choice of the kind of abstractions to be looked for. This notion is compatible with the idea that, in the process of development of a method for the identification of reusable assets, it is important to decide before what is wanted to be looked for in code and next to tailor a method in order to identify in the code the kind of abstraction required.

The set of approaches proposed can be grouped in two main families. The first family is composed of the methods based on the global variables, on the user types of formal parameters in procedures, on the data bounding and so on. All these methods are looking for data abstraction. In the second family the approaches are based on the dominance relations, on the slicing, on testing and so on. All these methods are looking for functional abstractions. In the course of the description of each method some reflections on the way they work of it has been presented and, in some cases, experimental results have been presented.

The description of the methods has been completed with one approach based on metrics that is useful for the evaluation of the quality of the components extracted and proposed for reuse. Because the quality of the modules depends on the quality of the method adopted, then good quality of the modules imply good quality of the method in extracting abstractions of the same kind of the modules extracted. Therefore, also if not explicitely proposed, the method based on matrics can be used to evaluate the quality of a method for a specific kind

of abstraction.

• Development of a more precise method for the identification and extraction of reusable abstract data types

By analysing one of the existing methods for the extraction of reusable assets from existing code and based on the relationship existing between user-defined data types and the proceduer-like components that use them in the interface, some limitations of it were identified and it was realized that more satisfactory results could be obtained.

The analysis of the results obtained led to the development of a more precise method for the identification and the extraction of Abstract Data Types. For the kind of modules extracted the new method looks for data abstractions, but it uses some principles based on functional absrarctions to complete and re-organize the results obtained. In fact, the starting point of the new method is the cited method based on the USE relations between user-defined data types and procedure-like components. It looks for ADTs and manages in identifying the supporting structure and the operations of the extractable ADTs, but it does not identify all the procedure-like components necessary for the implementation of the operations. The new method overcomes this problem by using the *call*, *direct dominance* and *strongly direct dominance* relations. Through them it locates all the procedure-like components for the implementation of the operations of the ADTs and identify the USE relations existing between the modules. In the last phases it identifies the functional abstractions implemented in each ADT. Therefore, by looking both for data abstractions and for functional abstractions the new method givees more precise results than other approaches looking for abstractions of one kind.

• Formularization of the new method

The new method proposed has been formularized in a set of sets and relations by using the predicate calculus. This kind of formularization can synthetize all the summary relations existing between the software components and shows all the dependency relations existing between the summary relations. Moreover it offers an representation of the new method that can be translated easily in software code.

• Prototype implementation of the new method to show that it is automable

The ease to translate in software code the formal representation of the new method brought to the implementation of it by using a logical programming language like the Prolog. The

prototype tool available comprehends the tool implemented for the ADT method. By static analysis of the source code, this tool gives a collection of facts of arity 1 and arity 2 representing the direct relations of the software system. By combining the direct relations it is possible to obtain the information about the first structure of the identified ADTs. The new method tool queries the database of all those facts and combines them for the implementation of the relations formalized in the formal representation of the method and elaborates facts expressing the final structure of the identified ADTs and all the relations existing between the software components.

Appendix B shows the complete database of Prolog fact obtained by the execution of the tool with the program Editor.pas.

- **Establish a criterion for measuring the effectiveness of the new method**

In Chapter 5 it has been amply discussed the quality of the new method. The four quality attributes of *method completeness*, *adequacy*, *Purity* and *module completeness* have been evaluated. It was shown how better results have been obtained with reference to the results obtained on the same set of case studies experimented with the ADT method. It was discussed also that the new method can be assumed like a good instrument for the problem of the software comprehension. In fact it manages in dividing the full software system in more than one modules that, for their dimensions, are easier to understand than the initial software system. Then the comprehension of a software system can happen through the comprehension of the single modules and the study of their interactions.

6.3 Further Work

Following on from the work showed in this thesis, there are a number of additional researches that can be interesting to investigate.

One of them arises from the consideration that the starting point of the new approach is a method based on the relations existing between the user-defined data type and the procedure-like components that use them in the interface to declare formal parameters and/or return value in the function. It does not take into account the case in which the exchange of information between procedure-like components happens not through formal parameters but through global variables. This is the case of the program Format.pas. When compared with the other programs it gave poor results. It was splitted in modules of smaller dimensions

but they appeared quite complex and not certainly implementing Abstract Data Types. The program does a big use of global variables and it is supposed that by considering them it is possible to obtain better results. Then the next studies will be involved with the analysis of the behaviour of the global variables when the procedure-like component are grouped and considered like in the new method. To reach that aim the relations existing between user-defined data types and global variables will be considered and dominance and use relations between variables will be also considered.

Another field of study will be represented by the experimentation of the new method with programming languages different from Pascal. It is supposed that the new method gives good results also with the 'C' programming language, but real case studies are not available in this moment for the difficulties met in finding 'C' software of suitable dimension using user-defined data types. It will be interesting to analyse the behaviour of the new method with programming language like Cobol in which it is not possible to define software components of the kind procedure-like.

Appendix A

Notation Adopted

The notation adopted for the formulation of the new method is in the form of sets and relations and derives from Ince [41]. In this notation, the way of defining a set is known as a *comprehensive specification*, and it enables a set to be succintly and unambiguously defined. Its general form is:

$$\{< signature > \mid < predicate > \bullet < term >\}.$$

where *signature* consists of a series of identifiers together with the set to which they belong. An example of a signature is '$x : A$' that establish that x is an element of the set A. The *predicate* part defines the properties of the members of the set to define; it is expressed by following the rules of *predicate calculus* by using logic and relational]operators and the existential and universal quantifiers. The *term* part defines the form of the member of the set. Examples of sets are:

$$\{n : N \mid n > 20 \ \wedge \ n < 100 \ \bullet \ n\} \qquad \{x, y : N \mid x + y = 5 \ \bullet \ x^2 + y^2\}$$

The first specifies the set of natural numbers which satisfy the condition $n > 20 \ \wedge \ n < 100$; the second is the set of natural numbers of the form $x^2 + y^2$ where $x + y$ equals 5, that is the set $\{13, 17, 25\}$.

This kind of notation can also be used to represent a *relation*. A relation is used to express the fact that there is a connection between the elements, belonging to some sets, that make up an ordered pair. In the constructive specification of a relation it is the *predicate* which defines this connection, while the *term* part appears like a couple of elements. For example, the writing

$\{a : A, b : B \mid p(a, b) \bullet (a, b)\}$

expresses a relation whose elements are the couples (a, b), with a belonging to A and b belonging to B, defined from the predicate $p(a, b)$.

When the ordered pairs that makeup a relation are obtained from two sets A and B it is usual to refer to the relation as defined on AxB. The fact that an ordered pair (a, b) is contained in a relation R can be written as $(a, b) \in R$ or $R(a, b)$. The reference to a particular element of a relation will be written in one of these two ways.

A number of operators are defined for relations. Two of them are the domain operator *dom* and the range operator *rng*. The first operator defines the set whose members are the left-hand elements of the pairs in a relation, and, given a relation R over $T_1 x T_2$, it is defined as:

$$dom_R = \{t_1 : T_1 \mid \exists\, t_2 : T_2 \bullet R(t_1, t_2)\}.$$

The second operator is similar to *dom*. It returns the set of the right-hand elements in a relation. Its definition for the relation R is the following:

$$rng_R = \{t_2 : T_2 \mid \exists\, t_1 : T_1 \bullet R(t_1, t_2)\}.$$

Other interesting operators are the operators of *restriction* of the domain and range of a relation. The operator to restrict the domain, identified by the symbol $\lhd$, has two operands: the first is a set; the second is a relation. It forms a subset of of the second operand which only contains pairs whose first elements are contained in the first operand. Given a relation R over $T_1 x T_2$, and a subset S_1 of T_1, the relation restricted of R on S_1 is defined as follows:

$$S_1 \lhd R = \{t_1 : T_1, t_2 : T_2 \mid t_1 \in S_1 \bullet (t_1, t_2)\}.$$

In a similar way the operator, $\rhd$, a restriction of the range is defined. Given a relation R over $T_1 x T_2$, and a subset S_2 of T_2, the relation restricted of R on S_2 is defined as:

$$R \rhd S_2 = \{t_1 : T_1, t_2 : T_2 \mid t_2 \in S_2 \bullet (t_1, t_2)\}.$$

Appendix B

In the following the complete collections of direct and summary relations obtained by the application of the tool on the program Editor.pas is shown. The facts expressing the direct relations are produced by a static analyser written in Lex and Yacc. They are followed by the collection of the summary relations obtained by applying the prototype tool written in Prolog and presented in Chapter 4. In this chapter the semantic of each fact was introduced also. It is referred to it for each explanation.

Direct Relations

```
user_def_type(linestring).
user_def_type(argstring).
user_def_type(linelength).
user_def_type(filenamestring).
user_def_type(patternstring).
user_def_type(tracestring).
user_def_type(lineptr).
user_def_type(linerec).
user_def_type(promptstring).
user_def_type(promptrec).

proc(getnew).
proc(etrace).
proc(readline).
proc(qrymem).
proc(filset).
proc(dumppat).
proc(catsub).
proc(linkup).
proc(freeline).
proc(clrbuf).
proc(setbuf).
proc(skipbl).
proc(optpat).
proc(getlst).
proc(docmd).
proc(ckglob).
proc(doglob).
```

```
proc(initialize).

func(intrpt).
func(readcmd).
func(readterm).
func(assignfile).
func(open).
func(ctoi).
func(addset).
func(esc).
func(patsiz).
func(locate).
func(omatch).
func(amatch).
func(match).
func(stclos).
func(getccl).
func(makpat).
func(maksub).
func(nextln).
func(prevln).
func(alloline).
func(getind).
func(getpak).
func(gettxt).
func(injpak).
func(inject).
func(append).
func(delete).
func(ckp).
func(defalt).
func(getfn).
func(ptscan).
func(getnum).
func(getone).
func(getrhs).
func(dolist).
func(doprnt).
func(doread).
func(dowrit).
func(move).
func(subst).

proc_func_dec(main,getnew).
proc_func_dec(main,etrace).
proc_func_dec(main,readline).
proc_func_dec(main,intrpt).
proc_func_dec(main,qrymem).
proc_func_dec(main,readcmd).
proc_func_dec(readcmd,readterm).
proc_func_dec(main,assignfile).
proc_func_dec(main,open).
proc_func_dec(main,ctoi).
proc_func_dec(main,addset).
```

UNIV DUNELM

```
proc_func_dec(main,esc).
proc_func_dec(main,filset).
proc_func_dec(main,dumppat).
proc_func_dec(main,patsiz).
proc_func_dec(main,locate).
proc_func_dec(main,omatch).
proc_func_dec(main,amatch).
proc_func_dec(main,match).
proc_func_dec(main,stclos).
proc_func_dec(main,getccl).
proc_func_dec(main,makpat).
proc_func_dec(main,catsub).
proc_func_dec(main,maksub).
proc_func_dec(main,nextln).
proc_func_dec(main,prevln).
proc_func_dec(main,linkup).
proc_func_dec(main,alloline).
proc_func_dec(main,freeline).
proc_func_dec(main,getind).
proc_func_dec(main,getpak).
proc_func_dec(main,gettxt).
proc_func_dec(main,injpak).
proc_func_dec(main,inject).
proc_func_dec(main,append).
proc_func_dec(main,delete).
proc_func_dec(main,clrbuf).
proc_func_dec(main,setbuf).
proc_func_dec(main,ckp).
proc_func_dec(main,defalt).
proc_func_dec(main,skipbl).
proc_func_dec(main,getfn).
proc_func_dec(main,ptscan).
proc_func_dec(main,optpat).
proc_func_dec(main,getnum).
proc_func_dec(main,getone).
proc_func_dec(main,getlst).
proc_func_dec(main,getrhs).
proc_func_dec(main,dolist).
proc_func_dec(main,doprnt).
proc_func_dec(main,doread).
proc_func_dec(main,dowrit).
proc_func_dec(main,move).
proc_func_dec(main,subst).
proc_func_dec(main,docmd).
proc_func_dec(main,ckglob).
proc_func_dec(main,doglob).
proc_func_dec(main,initialize).

proc_use_type_in_interface(getnew,lineptr).
proc_use_type_in_interface(etrace,tracestring).
proc_use_type_in_interface(readline,linestring).
proc_use_type_in_interface(filset,linestring).
proc_use_type_in_interface(dumppat,patternstring).
proc_use_type_in_interface(catsub,linestring).
```

```
proc_use_type_in_interface(catsub,patternstring).
proc_use_type_in_interface(linkup,lineptr).
proc_use_type_in_interface(freeline,lineptr).

func_use_type_in_interface(readcmd,linestring).
func_use_type_in_interface(readterm,linestring).
func_use_type_in_interface(readterm,promptrec).
func_use_type_in_interface(assignfile,filenamestring).
func_use_type_in_interface(open,filenamestring).
func_use_type_in_interface(ctoi,linestring).
func_use_type_in_interface(addset,linestring).
func_use_type_in_interface(esc,linestring).
func_use_type_in_interface(patsiz,patternstring).
func_use_type_in_interface(locate,patternstring).
func_use_type_in_interface(omatch,linestring).
func_use_type_in_interface(omatch,patternstring).
func_use_type_in_interface(amatch,linestring).
func_use_type_in_interface(amatch,patternstring).
func_use_type_in_interface(match,linestring).
func_use_type_in_interface(match,patternstring).
func_use_type_in_interface(stclos,patternstring).
func_use_type_in_interface(getccl,argstring).
func_use_type_in_interface(getccl,patternstring).
func_use_type_in_interface(makpat,argstring).
func_use_type_in_interface(makpat,patternstring).
func_use_type_in_interface(maksub,argstring).
func_use_type_in_interface(maksub,patternstring).
func_use_type_in_interface(alloline,lineptr).
func_use_type_in_interface(getind,lineptr).
func_use_type_in_interface(getpak,lineptr).
func_use_type_in_interface(gettxt,lineptr).
func_use_type_in_interface(inject,linestring).
func_use_type_in_interface(getfn,filenamestring).
func_use_type_in_interface(getrhs,patternstring).
func_use_type_in_interface(doread,filenamestring).
func_use_type_in_interface(dowrit,filenamestring).
func_use_type_in_interface(subst,patternstring).

used_to_define(linestring,argstring).
used_to_define(linestring,patternstring).
used_to_define(lineptr,linerec).
used_to_define(linelength,linerec).
used_to_define(linestring,linerec).
used_to_define(promptstring,promptrec).

proc_func_call(readcmd,readterm).
proc_func_call(open,assignfile).
proc_func_call(filset,addset).
proc_func_call(filset,esc).
proc_func_call(omatch,locate).
proc_func_call(amatch,omatch).
proc_func_call(amatch,patsiz).
proc_func_call(match,amatch).
proc_func_call(stclos,addset).
```

```
proc_func_call(getccl,addset).
proc_func_call(getccl,filset).
proc_func_call(makpat,addset).
proc_func_call(makpat,getccl).
proc_func_call(makpat,stclos).
proc_func_call(makpat,esc).
proc_func_call(catsub,addset).
proc_func_call(maksub,addset).
proc_func_call(maksub,esc).
proc_func_call(alloline,getnew).
proc_func_call(getpak,getind).
proc_func_call(gettxt,getpak).
proc_func_call(injpak,getind).
proc_func_call(injpak,alloline).
proc_func_call(injpak,linkup).
proc_func_call(inject,injpak).
proc_func_call(append,etrace).
proc_func_call(append,readcmd).
proc_func_call(append,inject).
proc_func_call(delete,prevln).
proc_func_call(delete,getind).
proc_func_call(delete,intrpt).
proc_func_call(delete,linkup).
proc_func_call(delete,freeline).
proc_func_call(setbuf,alloline).
proc_func_call(setbuf,delete).
proc_func_call(ckp,etrace).
proc_func_call(defalt,etrace).
proc_func_call(getfn,etrace).
proc_func_call(getfn,skipbl).
proc_func_call(ptscan,etrace).
proc_func_call(ptscan,intrpt).
proc_func_call(ptscan,nextln).
proc_func_call(ptscan,prevln).
proc_func_call(ptscan,gettxt).
proc_func_call(ptscan,match).
proc_func_call(optpat,etrace).
proc_func_call(optpat,makpat).
proc_func_call(getnum,etrace).
proc_func_call(getnum,ctoi).
proc_func_call(getnum,optpat).
proc_func_call(getnum,ptscan).
proc_func_call(getone,etrace).
proc_func_call(getone,skipbl).
proc_func_call(getone,getnum).
proc_func_call(getlst,etrace).
proc_func_call(getlst,getone).
proc_func_call(getrhs,etrace).
proc_func_call(getrhs,maksub).
proc_func_call(dolist,intrpt).
proc_func_call(dolist,gettxt).
proc_func_call(doprnt,intrpt).
proc_func_call(doprnt,gettxt).
proc_func_call(doread,open).
```

```
proc_func_call(doread,intrpt).
proc_func_call(doread,readline).
proc_func_call(doread,injpak).
proc_func_call(dowrit,open).
proc_func_call(dowrit,intrpt).
proc_func_call(dowrit,getpak).
proc_func_call(move,getind).
proc_func_call(move,prevln).
proc_func_call(move,nextln).
proc_func_call(move,linkup).
proc_func_call(subst,etrace).
proc_func_call(subst,intrpt).
proc_func_call(subst,gettxt).
proc_func_call(subst,amatch).
proc_func_call(subst,catsub).
proc_func_call(subst,addset).
proc_func_call(subst,delete).
proc_func_call(subst,inject).
proc_func_call(docmd,etrace).
proc_func_call(docmd,append).
proc_func_call(docmd,defalt).
proc_func_call(docmd,delete).
proc_func_call(docmd,prevln).
proc_func_call(docmd,ckp).
proc_func_call(docmd,nextln).
proc_func_call(docmd,getone).
proc_func_call(docmd,move).
proc_func_call(docmd,optpat).
proc_func_call(docmd,getrhs).
proc_func_call(docmd,subst).
proc_func_call(docmd,getfn).
proc_func_call(docmd,clrbuf).
proc_func_call(docmd,setbuf).
proc_func_call(docmd,doread).
proc_func_call(docmd,dowrit).
proc_func_call(docmd,doprnt).
proc_func_call(docmd,dolist).
proc_func_call(docmd,skipbl).
proc_func_call(ckglob,etrace).
proc_func_call(ckglob,optpat).
proc_func_call(ckglob,defalt).
proc_func_call(ckglob,intrpt).
proc_func_call(ckglob,gettxt).
proc_func_call(ckglob,match).
proc_func_call(ckglob,nextln).
proc_func_call(ckglob,getind).
proc_func_call(doglob,etrace).
proc_func_call(doglob,getind).
proc_func_call(doglob,intrpt).
proc_func_call(doglob,getlst).
proc_func_call(doglob,docmd).
proc_func_call(doglob,nextln).
proc_func_call(initialize,intrpt).
proc_func_call(main,initialize).
```

```
proc_func_call(main,setbuf).
proc_func_call(main,readcmd).
proc_func_call(main,etrace).
proc_func_call(main,getlst).
proc_func_call(main,ckglob).
proc_func_call(main,doglob).
proc_func_call(main,docmd).
proc_func_call(main,clrbuf).
```

Summary Relations

```
mod_type_comp(4, linestring)
mod_type_comp(5, argstring)
mod_type_comp(3, filenamestring)
mod_type_comp(5, patternstring)
mod_type_comp(1, tracestring)
mod_type_comp(2, lineptr)
```

```
mod_proc_comp(4, addset)
mod_proc_comp(4, ctoi)
mod_proc_comp(4, esc)
mod_proc_comp(4, filset)
mod_proc_comp(4, inject)
mod_proc_comp(4, readcmd)
mod_proc_comp(4, readline)
mod_proc_comp(4, readterm)
mod_proc_comp(3, assignfile)
mod_proc_comp(3, doread)
mod_proc_comp(3, dowrit)
mod_proc_comp(3, getfn)
mod_proc_comp(3, open)
mod_proc_comp(5, amatch)
mod_proc_comp(5, catsub)
mod_proc_comp(5, dumppat)
mod_proc_comp(5, getccl)
mod_proc_comp(5, getrhs)
mod_proc_comp(5, locate)
mod_proc_comp(5, makpat)
mod_proc_comp(5, maksub)
mod_proc_comp(5, match)
mod_proc_comp(5, omatch)
mod_proc_comp(5, patsiz)
mod_proc_comp(5, stclos)
mod_proc_comp(5, subst)
mod_proc_comp(1, etrace)
mod_proc_comp(2, alloline)
mod_proc_comp(2, freeline)
mod_proc_comp(2, getind)
mod_proc_comp(2, getnew)
```

```
dir_dom_comp(main)
dir_dom_comp(open)


call_dir_dom_comp(4, addset)
call_dir_dom_comp(4, esc)
call_dir_dom_comp(4, injpak)
call_dir_dom_comp(5, addset)
call_dir_dom_comp(5, amatch)
call_dir_dom_comp(5, esc)
call_dir_dom_comp(5, etrace)
call_dir_dom_comp(5, intrpt)
call_dir_dom_comp(3, etrace)
call_dir_dom_comp(3, getpak)
call_dir_dom_comp(3, injpak)
call_dir_dom_comp(3, intrpt)
call_dir_dom_comp(3, open)
call_dir_dom_comp(2, addset)
call_dir_dom_comp(2, amatch)
call_dir_dom_comp(2, esc)
call_dir_dom_comp(2, etrace)
call_dir_dom_comp(2, intrpt)
call_dir_dom_comp(1, getind)
call_dir_dom_comp(1, getpak)
call_dir_dom_comp(2, getind)
call_dir_dom_comp(2, getpak)
call_dir_dom_comp(delete, getind)
call_dir_dom_comp(delete, intrpt)
call_dir_dom_comp(delete, linkup)
call_dir_dom_comp(injpak, getind)
call_dir_dom_comp(injpak, linkup)
call_dir_dom_comp(main, etrace)


mod_call_proc(4, addset)
mod_call_proc(4, esc)
mod_call_proc(4, getind)
mod_call_proc(4, injpak)
mod_call_proc(4, linkup)
mod_call_proc(5, addset)
mod_call_proc(5, amatch)
mod_call_proc(5, esc)
mod_call_proc(5, etrace)
mod_call_proc(5, getind)
mod_call_proc(5, intrpt)
mod_call_proc(5, linkup)
mod_call_proc(3, etrace)
mod_call_proc(3, getind)
mod_call_proc(3, getpak)
mod_call_proc(3, injpak)
mod_call_proc(3, intrpt)
mod_call_proc(3, linkup)
mod_call_proc(3, open)
mod_call_proc(2, addset)
```

```
mod_proc_comp(2, getpak)
mod_proc_comp(2, gettxt)
mod_proc_comp(2, linkup)


dom_proc(injpak, 1, alloline)
dom_proc(open, 1, assignfile)
dom_proc(subst, 1, catsub)
dom_proc(main, 1, ctoi)
dom_proc(subst, 1, delete)
dom_proc(main, 1, doread)
dom_proc(main, 1, dowrit)
dom_proc(getccl, 1, filset)
dom_proc(delete, 1, freeline)
dom_proc(makpat, 1, getccl)
dom_proc(main, 1, getfn)
dom_proc(alloline, 1, getnew)
dom_proc(main, 1, getrhs)
dom_proc(subst, 1, gettxt)
dom_proc(subst, 1, inject)
dom_proc(omatch, 1, locate)
dom_proc(main, 1, makpat)
dom_proc(getrhs, 1, maksub)
dom_proc(main, 1, match)
dom_proc(amatch, 1, omatch)
dom_proc(amatch, 1, patsiz)
dom_proc(delete, 1, prevln)
dom_proc(main, 1, readcmd)
dom_proc(doread, 1, readline)
dom_proc(readcmd, 1, readterm)
dom_proc(getfn, 1, skipbl)
dom_proc(makpat, 1, stclos)
dom_proc(main, 1, subst)
dom_proc(main, 2, amatch)
dom_proc(main, 2, etrace)
dom_proc(main, 2, open)
dom_proc(main, 3, getpak)
dom_proc(main, 3, injpak)
dom_proc(main, 3, intrpt)
dom_proc(main, 4, addset)
dom_proc(main, 4, esc)
dom_proc(main, 4, getind)
dom_proc(main, 4, linkup)


dir_dom_comp(addset)
dir_dom_comp(amatch)
dir_dom_comp(esc)
dir_dom_comp(etrace)
dir_dom_comp(getind)
dir_dom_comp(getpak)
dir_dom_comp(injpak)
dir_dom_comp(intrpt)
dir_dom_comp(linkup)
```

```
mod_call_proc(2, amatch)
mod_call_proc(2, esc)
mod_call_proc(2, etrace)
mod_call_proc(2, getind)
mod_call_proc(2, intrpt)
mod_call_proc(2, linkup)
mod_call_proc(1, getind)
mod_call_proc(1, getpak)
mod_call_proc(2, getpak)


mod_def_proc(4, addset)
mod_def_proc(4, ctoi)
mod_def_proc(4, esc)
mod_def_proc(4, filset)
mod_def_proc(4, inject)
mod_def_proc(4, readcmd)
mod_def_proc(4, readline)
mod_def_proc(4, readterm)
mod_def_proc(3, assignfile)
mod_def_proc(3, doread)
mod_def_proc(3, dowrit)
mod_def_proc(3, getfn)
mod_def_proc(3, open)
mod_def_proc(5, amatch)
mod_def_proc(5, catsub)
mod_def_proc(5, dumppat)
mod_def_proc(5, locate)
mod_def_proc(5, makpat)
mod_def_proc(5, maksub)
mod_def_proc(5, match)
mod_def_proc(5, omatch)
mod_def_proc(5, patsiz)
mod_def_proc(5, stclos)
mod_def_proc(5, subst)
mod_def_proc(1, etrace)
mod_def_proc(2, alloline)
mod_def_proc(2, freeline)
mod_def_proc(2, getind)
mod_def_proc(2, getnew)
mod_def_proc(2, getpak)
mod_def_proc(2, gettxt)
mod_def_proc(2, linkup)
mod_def_proc(5, getccl)
mod_def_proc(5, getrhs)
mod_def_proc(4, injpak)
mod_def_proc(5, delete)
mod_def_proc(5, intrpt)
mod_def_proc(3, injpak)
mod_def_proc(3, intrpt)
mod_def_proc(3, skipbl)
mod_def_proc(5, prevln)
```

```
mod_def_type(2, linestring)
mod_def_type(5, argstring)
mod_def_type(3, filenamestring)
mod_def_type(5, patternstring)
mod_def_type(1, tracestring)
mod_def_type(2, lineptr)
mod_def_type(2, linelenght)
mod_def_type(2, linestring)
mod_def_type(4, linestring)
mod_def_type(2, linerec)


mod_exp_proc(1, etrace)
mod_exp_proc(2, alloline)
mod_exp_proc(2, getind)
mod_exp_proc(2, getpak)
mod_exp_proc(2, gettxt)
mod_exp_proc(2, linkup)
mod_exp_proc(2, alloline)
mod_exp_proc(2, freeline)
mod_exp_proc(3, doread)
mod_exp_proc(3, dowrit)
mod_exp_proc(3, getfn)
mod_exp_proc(3, open)
mod_exp_proc(4, addset)
mod_exp_proc(4, ctoi)
mod_exp_proc(4, esc)
mod_exp_proc(4, filset)
mod_exp_proc(4, inject)
mod_exp_proc(4, readcmd)
mod_exp_proc(4, readline)
mod_exp_proc(5, amatch)
mod_exp_proc(5, getrhs)
mod_exp_proc(5, makpat)
mod_exp_proc(5, match)
mod_exp_proc(5, subst)
mod_exp_proc(5, dumppat)


mod_exp_type(1, tracestring)
mod_exp_type(2, lineptr)
mod_exp_type(3, filenamestring)
mod_exp_type(3, statusrange)
mod_exp_type(4, linestring)
mod_exp_type(5, argstring)
mod_exp_type(5, patternstring)


proc_dec_proc(open, assignfile)
proc_dec_proc(doread, injpak)
proc_dec_proc(getfn,skipl)
proc_dec_proc(readcmd, readterm)
proc_dec_proc(inject, injpak)
proc_dec_proc(amatch, omatch)
```

```
proc_dec_proc(amatch, patsiz)
proc_dec_proc(omatch, locate)
proc_dec_proc(makpat, getccl)
proc_dec_proc(makpat, stclos)
proc_dec_proc(subst, delete)
proc_dec_proc(subst, catsub)
proc_dec_proc(getrhs, maksub)


mod_uses(3, 1)
mod_uses(3, 2)
mod_uses(3, 4)
mod_uses(4, 2)
mod_uses(5, 1)
mod_uses(5, 2)
mod_uses(5, 4)
```

Bibliography

[1] Software Engineering, **Sommerville I.**, *Addison-Wesley Publishing Company*, 1989

[2] Software Engineering, **Boehm B.W.**, *IEEE Transactions on Computer, 25(2)*, 1976, pp.1226-1241

[3] Royce W.W., **Managing the development of large software systems**, *Proc. WESTCOM San Francisco CA*, 1970

[4] -, **Software Engineering Standard**, *ANSI/IEEE std729*, 1983

[5] -, **An American National Standard and IEEE Standard glossary of Software Engineering Terminology**, *IEEE Standard Boards and ANSI Standard Institute ANSI/IEEE Std610.12*, 1990

[6] Lientz B.P. and Swanson E.B., **Software Maintenance Management**, *Addison-Wesley Reading MA*, 1980

[7] Hall P.A.V., **Software reuse, reverse engineering, and re-engineering**, *Software Reuse and Reverse Engineering in Practice, UNICOM, Applied Information Technology 12*, 1992, pp.3-31

[8] Jones R., **Business Software Review**, *jan/feb 1988*,

[9] Biggerstaff T., and Perlis A., **Software Reusability**, *ACM Press*, 1989

[10] Canfora, G., Cimitile, A., and Munro, M., **RE^2: Reverse Engineering and Reuse Re-Engineering**, *Journal of Software Maintenance, Research and Practice, Wiley*, 1994

[11] -, **Reference Manual for the Ada Programming Language**, *US Dep. Defence, MIL STD 1815A*, 1983

[12] Wirth, N., **Programming in Modula-2**, *Springer-Verlag, New York, Third corrected edition*, 1985

[13] Antonini, P., Benedusi, P., Cantone, G., and Cimitile, A., **Maintenance and Reverse Engineering: Low Level Design Documents Production and Improvement**, *Proc. of Conference on Software Maintenance, Austin, Texas, IEEE Comp. Soc. Press*, 1987, pp.91-100

[14] Cimitile, A., and Visaggio, G., **Software Salvaging and the Dominance Tree**, *Journal of Systems and Software, vol.2 no.1*, Feb. 1995

[15] Cimitile, A., Fasolino, A.R., and Maresca, P., **Reuse Reengineering and Validation via Concept Assignment**, *Proc. of Conference on Software Maintenance, Montreal, Canada, IEEE Comp. Soc. Press, 1993*, Oct. 1993, pp.216-225

[16] Hetch M.S., **Flow Analysis of Computer Programs**, *Elsevier North-Holland*, 1977

[17] Liu S.S., and Wilde N., **Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery**, *Proc. of Conference on Software Maintenance, San Diego, California, USA, IEEE Comp. Soc. Press*, Nov. 26-29 1990, pp.266-271

[18] Dunn M.F., and Knight J.C., **Automating the Detection of Reusable Parts in Existing Software**, *Proc. of International Conference on Software Engineering, Baltimore, Maryland, IEEE Comp. Soc. Press*, 1993, pp.381-390

[19] Livadas P.E., and Roy P.K., **Program Dependence Analysis**, *Proc. of IEEE Conference on Software Maintenance, Orlando, Florida, IEEE Comp. Soc. Press*, Nov. 9-12 1992

[20] Haughton H.P., and Lano K., **Object Revisited**, *Proc. of Conference on Software Maintenance CSM'91, IEEE Comp. Soc. Press*, 1991

[21] Lano K., **A Semantics for an Object Oriented Specification Language**, *2487-TN-PRG-1045 Oct.*, 1990

[22] Canfora, G., Cimitile, A., and Munro, M., **A Reverse Engineering Method for Identifying Reusable Abstract Data Types**, *Proc. of Working Conference on Reverse Engineering, Maryland, IEEE Comp. Soc. Press*, May 1993, pp.73-82

[23] Canfora, G., Cimitile, A., Munro, M., and Tortorella, M., **Experiments in Identifying Reusable Abstract Data Types in Program Code**, *Proc. of 2-nd Workshop on Program Comprehension, Capri, Italy, IEEE Comp. Soc. Press*, 1993, pp.36-45

[24] Canfora, G., Cimitile, A., Munro, M., and Taylor, C.J., **Extracting Abstract Data Types from C Programs: A Case Study**, *Proc. of Conference on Software Maintenance, Montreal, Canada, IEEE Comp. Soc. Press*, Oct. 1993, pp. 200-209

[25] Ning J.Q., Engberts A., and Kozaczynsky W., **Recovering Reusable Components from Legacy Systems by Program Segmentation**, *Proc. of Working Conference on Reverse Engineering, Maryland, IEEE Comp. Soc. Press*, May 1993, pp.64-72

[26] Cutillo F., Fiore P., and Visaggio G., **Identification and Extraction of "Domain Independent" Components in Large Program**, *Proc. of Working Conference on Reverse Engineering, Maryland, IEEE Comp. Soc. Press*, May 1993, pp.83-91

[27] Weiser M., **Program Slicing**, *IEEE Trans. on Software Engineering, vol. 10, n. 4 july 1984*, 1984

[28] Wilde N., Gomez J.A., Gust T., and Strasburg D., **Locating User Functionality in Old Code**, *IEEE, pp.200-205*, 1993

[29] Schwanke R.W., **An Intelligent Tool For Re-engineering Software Modularity**, *Proc. of 13th Int. Conf. on Software Engineering*, 1991, pp.83-92

[30] Parnas D.L., **Information Distribution Aspects of Design Methology**, *Information Processing 71, North-Holland Publishing Company*, 1972

[31] Hutchens D.H.., Basili V.R., **System Structure Analysis: Clustering with Data Binding**, *IEEE Trans. on Software Engineering, vol. SE-11, n. 8 august 1985, pp. 749-757*, 1985

[32] Delis A., and Basili V.R., **Data Binding Tool: a Toolfor Measurement Based on Data and Type Binding**, *PhD Thesis, University of Mariland*, 1990

[33] De Lucia, A., Di Lucca, G.A., and Fasolino A.R., **Towards the Evaluation of Reengineering Effort to Reuse Existing Software**, *International Conference on Achieving Quality in Software*, 1993

[34] Calliss, F.W., **Inter-module Code Analysis Techniques for Software Maintenance**, *PhD Thesis, University of Durham*, 1989

[35] Canfora, G., Cimitile, A., Munro, M., and Tortorella, M., **A Precise Method for Identifying Reusable Abstract Data Types in Code**, *Proc. of International Conference on Software Maintenance, Victoria, Canada, IEEE Comp. Soc. Press*, 19-23 Sept. 1994, pp.404-413

[36] Canfora, G., Cimitile, A., and Visaggio, G., **Assessing modularisation and code scavenging techniques**, *to appear on the Journal of Software Maintenance, Research and Practice, Wiley*, 1994

[37] Cimitile, A., Munro, M., and Tortorella, M., **Program Comprehension Through the Identification of Abstract Data Types**, *Proc. of 3-rd Workshop on Program Comprehension, Washington, U.S.A., IEEE Comp. Soc. Press*, 14-15 Nov. 1994

[38] Kernighan B., and Plauger P.J., **Software Tools in Pascal**, *Addison Wesley Publishing Company, Reading, Massachusetts*, 1981

[39] Kernighan B., and Pike R., **The Unix Programming Environment**, *Prentice Hall, Inc., Englewood Cliffs, New Jersey*, 1984

[40] Miller L.H., **Advanced Programming: Design and Structure Using Pascal**, *Addison Wesley Publishing Company, Reading, Massachusetts*, 1986

[41] Ince, D.C., **An Introduction to Discrete Mathematics and Formal System Specification**, *Clarendon Press - Oxford*, 1988