Design Recovery and Data Mining: A Methodology That Identifies Data-Cohesive Subsystems Based on Mining Association Rules.

Carlos Montes de oca

Louisiana State University and Agricultural & Mechanical College

Follow this and additional works at: https://digitalcommons.lsu.edu/gradschool_disstheses

Recommended Citation

https://digitalcommons.lsu.edu/gradschool_disstheses/6949

This Dissertation is brought to you for free and open access by the Graduate School at LSU Digital Commons. It has been accepted for inclusion in LSU Historical Dissertations and Theses by an authorized administrator of LSU Digital Commons. For more information, please contact gradetd@lsu.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
DESIGN RECOVERY AND DATA MINING:
A METHODOLOGY THAT IDENTIFIES DATA COHESIVE SUBSYSTEMS
BASED ON MINING ASSOCIATION RULES

A Dissertation
Submitted to the Graduate Faculty of the
Louisiana State University and
Agricultural and Mechanical College
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

in

The Department of Computer Science

by
Carlos Montes de Oca
B.S., Instituto Tecnológico y de Estudios Superiores de Monterrey, 1986
M.S., Universidad Autónoma Metropolitana, 1993
May 1999
To Lolita, Frida, ISAbel, César and Carlos

with love and gratitude
because we are one
yet we are many
because this text was written by one
but it was the work of many

Gracias, por todo
ACKNOWLEDGMENTS

I could not have done this dissertation without the help of many persons who directly or indirectly contributed to this work. In particular, I want to acknowledge the academic, financial, and emotional help that I received from the following persons and organizations.

First, I want to express my thankfulness to my advisor Dr. Doris L. Carver. Her help, support, and advice were essential for the completion of this research. Especially, I want to thank Dr. Carver for being a caring mentor. She always helped me to find a solution not only to academic issues but also to financial and legal issues derived from my alien status in the United States of America.

In addition, I want to thank the members of my research committee for their guidance and support, and to Michael A. Hoffman, Joel L. Donnelly, and César Montes de Oca for giving me access to real-world software systems that allowed me to test and fine tune the methodology produced in this research.

I want to express my gratitude to the Conacyt (National Council for Science and Technology of Mexico) and the Fulbright program for their financial support, and to the Department of Computer Science of Louisiana State University for granting me a graduate student assistantship.

Finally, I want to thank my siblings, Frida and César, and my parents, Lolita and Carlos. Their support, guidance, and friendship were fundamental for the culmination of this work. I also want to thank my family-in-law, relatives, and friends for keeping me motivated. Last, but not least, I want to express my gratitude and love to Isabel. She has been a major source of motivation and support. I cannot imagine being here without her.
TABLE OF CONTENTS

ACKNOWLEDGMENTS ......................................................................................................... iii
LIST OF TABLES ..................................................................................................................... vi
LIST OF FIGURES ................................................................................................................ vii
ABSTRACT ............................................................................................................................... ix

CHAPTER 1. INTRODUCTION ................................................................................................ 1

CHAPTER 2. BACKGROUND ................................................................................................ 12
  2.1. Data Mining .................................................................................................................. 12
  2.2. Related Research ....................................................................................................... 19
    2.2.1 Design recovery ...................................................................................................... 20
    2.2.2 Subsystem classification ...................................................................................... 23
    2.2.3 Other related research ......................................................................................... 27
  2.3. Design Recovery and Data Mining ............................................................................. 30
    2.3.1 Motivation ........................................................................................................... 30
    2.3.2 The three-step method ....................................................................................... 33

CHAPTER 3. THE ISA METHODOLOGY ............................................................................ 36
  3.1. Introduction ................................................................................................................ 36
  3.2. Overview .................................................................................................................. 38
    3.2.1 The input ............................................................................................................ 40
    3.2.2 The output .......................................................................................................... 42
    3.2.3 The process ....................................................................................................... 46
  3.3. Build a Database View ............................................................................................ 52
    3.3.1 Parse the source code ....................................................................................... 52
    3.3.2 Assign unique identifications .......................................................................... 53
    3.3.3 Generate the alpha set ..................................................................................... 54
  3.4. Perform Data Mining ............................................................................................... 55
  3.5. Consolidate and Interpret Results ........................................................................... 57
    3.5.1 The junta algorithm ......................................................................................... 58
    3.5.2 The assign-files algorithm ............................................................................... 70
    3.5.3 The form-hierarchy algorithm .......................................................................... 74
    3.5.4 The subsys-merger algorithm ......................................................................... 78
    3.5.5 The TO-ISA representation ............................................................................ 80

CHAPTER 4. THE REPRESENTATION MODEL ................................................................. 91
  4.1. Related Research ..................................................................................................... 91
  4.2. Description of RM ................................................................................................... 92
    4.2.1 Basic building blocks ....................................................................................... 93
    4.2.2 Additional notation ......................................................................................... 97
    4.2.3 Layers ............................................................................................................. 98
  4.3. Discussion .............................................................................................................. 102

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
CHAPTER 5. EVALUATION OF ISA ON REAL-WORLD SYSTEMS ........................................106
5.1. Automation of ISA ........................................................................................................107
  5.1.1 General description ......................................................................................................108
  5.1.2 Generation of the database view ................................................................................111
  5.1.3 Data mining ...............................................................................................................112
  5.1.4 Consolidation and presentation of the results ..............................................................113
5.2. Case Study: The TRS System ......................................................................................114
5.3. Case Study: The PS6 System .......................................................................................126
5.4. Discussion ......................................................................................................................137

CHAPTER 6. SUMMARY AND CONCLUSIONS ..............................................................141
6.1. Conclusions ...................................................................................................................142
6.2. Contributions ...............................................................................................................149
6.3. Future Work ..................................................................................................................151

REFERENCES .......................................................................................................................155

APPENDIX. TO-ISA REPORT FOR THE PS6 SYSTEM ..................................................161

VITA .......................................................................................................................................167
LIST OF TABLES

Table 1. Example of an input set to mine association rules .......................................................18
Table 2. Example of a context ...................................................................................................28
Table 3. ISA products ................................................................................................................46
Table 4. ISA products and their corresponding section in TO-ISA ..........................................82
Table 5. RM notation of the basic system components..............................................................93
Table 6. Programs and files in the TRS system ......................................................................122
Table 7. Programs, files, and subsystems for each main subsystem in PS6 ...........................130
Table 8. Metrics from the experiments on PS6........................................................................134
LIST OF FIGURES

Figure 1. Example of a concept lattice .......................................................................................29
Figure 2. Graphical representation of a subsystem decomposition ...........................................44
Figure 3. The steps of the ISA methodology ............................................................................47
Figure 4. Algorithm to produce the alpha set ............................................................................54
Figure 5. Header of the junta algorithm .....................................................................................60
Figure 6. Pseudocode of the first stage of the junta algorithm ...................................................62
Figure 7. Pseudocode of the is-member function ......................................................................66
Figure 8. Pseudocode of the second stage of the junta algorithm ..............................................67
Figure 9. Pseudocode of the merge procedure ...........................................................................68
Figure 10. Pseudocode of the is-subset function ......................................................................69
Figure 11. Example of transforming Z into S ...........................................................................70
Figure 12. Pseudocode of the Z-to-S algorithm ........................................................................71
Figure 13. Pseudocode of the assign-files algorithm ................................................................72
Figure 14. Pseudocode of the allocate function ........................................................................74
Figure 15. Header of the form-hierarchy algorithm ..................................................................75
Figure 16. Pseudocode of the form-hierarchy algorithm ...........................................................76
Figure 17. Pseudocode of the subsys-merger algorithm ............................................................79
Figure 18. Format of the Header section ....................................................................................83
Figure 19. Format of the Common Files section .......................................................................83
Figure 20. Format of the Subsystem section ..............................................................................85
Figure 21. Format of the Singular Programs and Unconnected Components sections ..........87
Figure 22. Format of the Supra Subsystems section ................................................................88
Figure 23. Example of a Supra Subsystems section with more than one hierarchy ...............88
Figure 24. RM notation for the hierarchies of file implications ..............................................94
ABSTRACT

Software maintenance is both a technical and an economic concern for organizations. Large software systems are difficult to maintain due to their intrinsic complexity, and their maintenance consumes between 50% and 90% of the cost of their complete life-cycle. An essential step in maintenance is reverse engineering, which focuses on understanding the system. This system understanding is critical to avoid the generation of undesired side effects during maintenance. The objective of this research is to investigate the potential of applying data mining to reverse engineering. This research was motivated by the following: (1) data mining can process large volumes of information, (2) data mining can elicit meaningful information without previous knowledge of the domain, (3) data mining can extract novel non-trivial relationships from a data set, and (4) data mining is automatable. These data mining features are used to help address the problem of understanding large legacy systems.

This research produced a general method to apply data mining to reverse engineering, and a methodology for design recovery, called Identification of Subsystems based on Associations (ISA). ISA uses mined association rules from a database view of the subject system to guide a clustering process that produces a data-cohesive hierarchical subsystem decomposition of the system. ISA promotes object-oriented principles because each identified subsystem consists of a set of data repositories and the code (i.e., programs) that manipulates them. ISA is an automatic multi-step process, which uses the source code of the subject system and multiple parameters as its input. ISA includes two representation models (i.e., text-based and graphic-based representation models) to present the resulting subsystem decomposition.

The automated environment RE-ISA implements the ISA methodology. RE-ISA was used to produce the subsystem decomposition of real-word software systems. Results show that ISA can automatically produce data-cohesive subsystem decompositions without previous
knowledge of the subject system, and that ISA always generates the same results if the same parameters are utilized.

This research provides evidence that data mining is a beneficial tool for reverse engineering and provides the foundation for defining methodologies that combine data mining and software maintenance.
CHAPTER 1. INTRODUCTION

Real-world software systems are under constant maintenance to adapt them to the dynamics of rapidly changing requirements. This incessant adaptation translates into large economic expenditures for organizations. Software reverse engineering and software maintenance are crucial areas of research because they study ways to cope with this problem. Real-world systems are large and complex. Therefore, reverse engineering and software maintenance require approaches capable of dealing with the size and complexity of actual software systems.

In this research, we define a new approach to reverse engineering and maintenance which is based on data mining. Data mining is a relatively new discipline focused on discovering relevant new knowledge from large databases. The motivation to pursue this new approach derives from the observation that data mining can discover unsuspected non-trivial relationships from large amounts of information. This observation suggests that data mining can be used to elicit new knowledge about the design of a subject system and that data mining can be applied to large software systems. These features make data mining a promising approach for coping with large and complex real-world systems. Consequently, the objective of this research is to show that data mining is a feasible and beneficial approach to reverse engineering and maintenance.

Chikofsky and Cross [Chik90] define reverse engineering as “the process of analyzing a software system to identify the system's components and their relationships, and to create representations of the system in another form or at a higher level of abstraction. Reverse engineering generally involves extracting design artifacts and building or synthesizing abstractions that are less implementation-dependent.” Therefore, the main objective of reverse engineering is to produce the information required to understand a software system. The comprehension of a software system is essential for maintenance and reengineering. Software
maintenance is concerned with modifying a software system to correct problems that appeared after delivery, to improve its current functionality or performance, and to adapt the system to the new conditions of its environment. In any of these activities, it is necessary to have a good understanding of the system before the maintenance task is performed to avoid problems such as side-effects, ripple-effects, lost functionality, degraded performance, loss of integrity, and loss of structure. Consequently, reverse engineering has a direct application to software maintenance. Reverse engineering tools also help to cope with complexity, to produce different views of the system, to recover lost information, and to facilitate reuse [Chik90].

Reverse engineering is not only of academic interest but also of an economic concern. In fact, there is an important economic justification for reverse engineering: the cost of maintenance. There are several estimates in the literature showing the economic impact of software maintenance, and even the most conservative estimate is troublesome. For example, Chikofsky [Chik90] indicates that maintenance accounts for 50 to 90\% of the total cost of the complete life-cycle of a software system. Ghezzi [Ghez91] indicates that the figure is 60\% or more, and other sources consider that the figure is greater than 70\% [Lien80] or greater than 75\% [Wood98]. In addition, the human resources invested in maintenance are large. Corbi [Corb89] indicates that 50\% to 80\% of the workload of programmers and managers is spent in maintenance related activities. Moreover, Corbi summarizes that about 50\% of the budget of a data processing unit goes to maintenance. Woods [Wood98] includes estimates that in 1990 the cost of software was about $100 billion dollars. If we consider that 75\% of that money was spent in maintenance, and that 80\% of the time spent in maintenance is devoted to understanding the system [Wood98], then we conclude that about $60 billion dollars were spent in understanding software in 1990 alone. Therefore, any technique that automates or improves current reverse engineering practices would have a significant economic impact.
Reverse engineering and maintenance are especially important in legacy systems. According to Bennett, legacy systems are “large software systems that we do not know how to cope with but that are vital to our organization” [Benn95]. Some of the characteristics of legacy systems include: they are old, they were written in early versions of third generation languages, they lack current documentation, they are large (e.g., in the range of hundreds of thousands of lines of source code), they have gone through several maintenance processes, they lack structure, and they are very difficult to understand. Moreover, there are few or no experts of the system because the original developers are gone from the organization. Nevertheless, as Bennett points out, legacy systems perform “crucial work for their organization,” they contain years of knowledge and experience about the business, they might be the only source of the business rules, they may be reliable, and they may be well adapted to the business environment. Thus, the problem with legacy systems is that the legacy system is a valuable asset for the organization, yet the legacy system becomes more expensive to maintain every day. As Bennett affirms, the legacy systems are “too expensive to maintain and the demands of the marketing department for alterations cannot be sustained. Business opportunities are being lost.” One of the options to cope with legacy systems is through reengineering, which addresses the problem in two stages. The first stage uses reverse engineering to understand the legacy system. The second stage, called forward engineering, uses the information produced in the reverse engineering stage and adds new specifications to rebuild the legacy system using modern technologies.

In addition to the area of reverse engineering, the area of data mining is a vital aspect of this research. Data mining is a specific part of the process of knowledge discovery in databases (KDD). The aim of this process is the extraction of useful unknown information from large volumes of data. Frawley defines knowledge discovery as “the nontrivial extraction of implicit, previously unknown, and potentially useful information from data [Fraw92]. For
example, the database that stores all the purchase information of a supermarket chain contains relevant information such as trends, patterns, and categorizations. However, this information is hidden somewhere in the large databases.

Some authors use the term data mining as a synonym for the KDD process. However, in this research we adopt the following definition of data mining [Fayy96a]: “a step in the KDD process consisting of particular data mining algorithms that, under some acceptable computational efficiency limitations, produces a particular enumeration of patterns \( E_j \) over \( F \),” where \( F \) is a data set of facts. A pattern “is an expression \( E \) in a language \( L \) describing facts in a subset \( F_E \) of \( F \). \( E \) is called a pattern if it is simpler than the enumeration of all facts in \( F_E \).” For example, a pattern could be “90% of the customers that buy product A in a visit buy product B in the next visit.” In short, data mining is the search of patterns in a data set. The search has to be done within certain acceptable performance limits. This definition does not restrict the type or form of the patterns that can be mined, suggesting that data mining can produce different types of “knowledge.” Some of the patterns that data mining algorithms mine are classifications, regressions, clusters, association rules, sequences, summarization, and dependency modeling.

In this research we combine reverse engineering and data mining. Reverse engineers and maintainers require techniques capable of extracting useful information from systems that are large, complex, poorly documented, and poorly understood (e.g., legacy systems). Data mining has some features that suggest that it can be used to address some of these problems. First, data mining can discover unsuspected non-trivial patterns and relationships among data elements in large databases. This feature can be used to address the problem of eliciting the relationships among system components. Second, data mining techniques can elicit relevant information without any previous knowledge of the object of study. This feature can be used to address the lack of proper documentation. Last, data mining can extract useful information
from large volumes of data, it is scalable, and it is automatable. Thus, data mining has the capability to deal with the complexity of the analysis of large software systems. These features of data mining suggest that data mining has the potential to produce the information (knowledge) required for reverse engineering and maintenance tasks.

These observations led to this research which explores the application of data mining technology to the areas of reverse engineering and maintenance. Hence, the research objective is to show that data mining is a feasible and beneficial approach to software reverse engineering and maintenance. This research involved both demonstrating that data mining can be applied to reverse engineering problems and showing the relevance of this approach. Consequently, the research hypothesis is: data mining techniques are feasible and beneficial tools for software reverse engineering.

We developed a three step approach to test the research hypothesis. The steps are: (1) develop a general method to apply data mining to reverse engineering, (2) develop a specific methodology, based on the general method and that uses data mining, to produce a reverse engineering artifact, and (3) evaluate the methodology with real-world software systems.

The first step of the approach is to develop a general method to apply data mining to reverse engineering. This method, called the three-step method (TS), consists of three general steps:

(1) Create a database representation of the target system. The database representation is derived from different sources of information such as documentation, domain knowledge, expert knowledge, and source code.

(2) Perform data mining on the database created in step 1. One or several data mining algorithms mine the database representation of the target system to elicit relevant knowledge about the system.
(3) Interpret and consolidate the mined patterns into a high level abstraction or artifact useful for reverse engineering and maintenance tasks.

The motivation for developing the TS method is that it can be used as a framework to define many methodologies to apply data mining to reverse engineering. For example, a specific methodology may store information on global variables and subprograms in the database, then it may mine association rules to produce an object-oriented design of the system. Thus, every methodology derived from the TS method should have a particular selection of the contents of its database, the data mining algorithms applied to the database, and the resulting reverse engineering artifact.

The next step of the research approach is to develop an instantiation of the three-step method to produce a specific reverse engineering product. The resulting instantiation is called the Identification of Subsystems using Associations (ISA) methodology. ISA instantiates the three-step method in the following way:

(1) ISA generates a database view of the system in the form of a table (i.e., a set of tuples).

(2) ISA mines the table looking for association rules. An association rule is an implication of the form "c% of the programs that use file X also use files Y and Z."

(3) ISA uses the mined associations to guide a clustering process that produces groups of programs that use a similar set of data files. Then, ISA merges the groups of programs to form a tree. Finally, it assigns files to each group of programs to form subsystems.

Therefore, ISA takes a software system composed of multiple programs and data files and organizes these components into a hierarchy of subsystems. A subsystem is defined as a subset of programs and data files. The input to the ISA methodology is the source code of the subject system, and the output is a report that describes the identified subsystems.
One important issue considered in the design of the ISA methodology was automation. Automation is a desired characteristic in reverse engineering techniques because the subject systems tend to be large; therefore, manual techniques are of limited practical application. In addition, one of the motivations of using data mining techniques for reverse engineering is precisely that data mining techniques are automatable. Thus, the development of ISA was oriented to profit from the automation potential of data mining.

In general terms, ISA is a design recovery technique. Chikofsky defines design recovery as “a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself” [Chik90]. Design recovery aims to produce high-level abstractions of the subject system to facilitate its understanding. Design recovery may use any available source of information such as source code, documentation, domain knowledge and personal experience. Similarly, design recovery may produce different results such as module breakdowns, structure charts, entity-relationship diagrams, and formal specifications. Design recovery also plays an important role in reuse by providing information that simplifies the localization of reusable parts. For example, current research on design recovery focus on identifying objects in legacy code. The idea is to use these objects to facilitate maintenance and promote reuse [Canf96].

ISA can be classified as a subsystem classification technique. Subsystem classification techniques, also called subsystem identification techniques, are a subset of design recovery. The objective of subsystem classification techniques is to produce an architectural description of the target system. Lakhota describes this area of design recovery in the following terms [Lakh97]: “An architecture of a software system classifies its components into subsystems and describes interactions between these subsystems. It provides a high-level abstraction of the
organization of a system and can be used to address the system level properties, such as capacity, throughput, consistency, and component capability.”

The information contained in an architectural description of a system is used in reverse engineering, maintenance, and reuse. Subsystem classification facilitates the analysis of the software system in several ways. It can be used to understand the general architecture of the system, to identify application domain concepts, to analyze the system at different abstraction levels, to split the analysis effort in teams, to evaluate maintenance activities, and to perform side-effect analysis. Subsystem classification also helps in the area of reengineering procedural programs into object oriented programs by helping to identify classes, methods, attributes, and the interrelationships among these elements. In reuse, subsystem identification can provide relevant information to detect reusable modules.

Specifically, ISA produces a data-cohesive hierarchical subsystem decomposition of the target system. ISA is divided into 11 steps and uses 9 parameters to control the subsystem decomposition process. The subsystem decomposition is data-cohesive because the identified subsystems are formed with programs that use a similar set of data files. The rationale behind this decomposition is that if a set of programs access the same data repositories, then the programs implement the processes that manipulate the data in the data files. Programs in a subsystem would implement similar or complementary functions on the data repositories assigned to the subsystem. Thus, ISA follows object-oriented principles because it promotes that the data and the code that manipulated the data are kept in a single unit. The subsystem decomposition is hierarchical because each subsystem may be decomposed in sub-subsystems, each sub-subsystem into sub-sub-systems, and so on until a primitive level is reached.

ISA produces other information besides the data cohesive hierarchical subsystem decomposition. For example, ISA identifies files that cannot be assigned to any particular subsystem, and programs that cannot be classified into any subsystem. In addition, a unique
outcome of ISA is the identification of hierarchies of file implications. A file implication is a relationship of the form “c% of the programs that use file f also use file g.” This file implication means that there is a c% confidence that every time file f is used also file g will be used. In other words, f implies g. A hierarchy of file implications may be used to summarize the files accessed by a program. For example, if a program p uses a file f in the file hierarchy, then p also uses all the ancestors of f in the hierarchy. The information that ISA produces also helps to highlight relationships among the system components. For example, it is easy to detect the files that are critical for each subsystem, the files that are used by several subsystems, and the files assigned to a particular subsystem but used in other subsystems. Moreover, ISA can classify the files assigned to a subsystem into files that are used exclusively inside the subsystem and those that also are used by external programs or subsystems.

ISA presents the results of the subsystem decomposition in two models, a text-based representation or textual output of ISA (TO-ISA), and a graph-based representation model (RM). The graphical representation model is particularly useful because the textual output may produce many pages of information, which makes the report difficult to read and to understand. RM is designed to graphically represent all the information produced by ISA. RM has notation to denote programs, files, subsystems, hierarchies of file implications, and different types of relationships among these components. In addition, RM uses layered diagrams to represent the hierarchy of subsystems. RM facilitates the identification of relevant information and relationships among system components. For example, with RM it is easy to locate the major components of each subsystem, the files that link several subsystems (critical resource), the main subsystems, and the major relationships among subsystems. Due to its layered structure, RM diagrams can be used to represent different levels of abstraction, from a
general map of the system to the particular description of a leaf subsystem in the subsystem decomposition tree.

Significant benefits of the ISA methodology are that it is automatable and that it does not require any previous knowledge of the subject system. The only source of information is the source code. This is an advantage because normally the only updated documentation besides the executable code is the source code. Therefore, ISA can be used to analyze large undocumented legacy systems.

The principal limitation of the ISA methodology is that it only can be used in systems that can be viewed as a collection of several relatively independent portions of code (i.e., programs) and several data repositories (i.e., data files). The programs and data files can be defined logically. For example, if all the source code of the system is in a single 500,000 lines file, it may be possible to identify several modules. In that case, those modules would become the "programs." However, if the system cannot be decomposed into several "programs" and "files," then ISA cannot be used.

The main contribution of this research is that it provides evidence that data mining is a beneficial tool for reverse engineering. Theoretical contributions of this research include the interpretation of an association rule in the context of reverse engineering, the definition of clustering techniques and metrics, the general three-step method to apply data mining to reverse engineering and maintenance, the ISA methodology and all the algorithms that implement it, and the definition of the graphical model RM. In addition, this research opens a promising research area: reverse engineering and data mining.

In this dissertation, we describe the major components of the research. Chapter 2 provides the background information. First, we describe the underlying ideas behind data mining and point to related literature. Next, Chapter 2 includes a related research section where the different approaches to design recovery and subsystem identification are discussed.
It explains the motivation for this research and describes the three-step method. Chapter 3 describes the ISA methodology. It lists and describes the general steps of the methodology. It includes a detailed description of all the algorithms used in each step as well as the definition of the TO-ISA representation. Chapter 4 defines the RM model, provides the motivations and design objectives for RM, and points to related literature. Chapter 5 describes the evaluation of the ISA methodology on real-world systems. It includes a description of the RE-ISA tool and describes its application to two software systems. In addition, chapter 5 includes a discussion and interpretation of the results of these experiments. Finally, Chapter 6 states the conclusions and contributions derived from this research, and lays the foundation for future work.
CHAPTER 2. BACKGROUND

In this chapter we present background concepts, review the literature of related approaches and techniques, and provide the motivation for this research. We present the underlying ideas behind data mining, its objectives, and its methods. We include a detailed description of the specific data mining technique used in this research. The related work section presents a review of different approaches to reverse engineering found in the literature with emphasis on design recovery. Specifically, this chapter includes a review of research on subsystem classification and a review of techniques used in reverse engineering that have some similarities to data mining techniques. Finally, we explain the motivation for this research and the proposed general method to exploit data mining in the reverse engineering and maintenance domains.

2.1. Data Mining

The process of extracting useful high-level information from large volumes of low-level data has different names such as data mining, knowledge discovery in databases, knowledge extraction, data archeology, and data pattern analysis. The interchangeable use of these terms can create confusion. The term data mining is the most widely used; however, in some research communities the term data mining is used to name a particular step of the process of extracting useful information from data. As Fayyad [Fayy96a] point out “the term knowledge discovery in databases, or KDD for short, was coined in 1989 to refer to the broad process of finding knowledge in data, and to emphasize the “high-level” application of particular data mining methods.” This dissertation adopts this convention of using KDD to refer to the overall knowledge extraction process and data mining to refer to one step in the process.

The KDD process was derived from the increasing need for tools to automatically analyze large volumes of data. For many years, computer systems have been accumulating data. Therefore, current databases are huge, and they are still growing rapidly. For example,
the US retailer Wal-Mart handles more than 20 million transactions per day [Fayy96a] and the NASA's Earth Observing System will produce several gigabytes per hour by the end of the century [Fayy96b]. Calculations indicate that the information in the world doubles every 20 months [Fraw92]. Hence, databases grow rapidly every day. The process of digging into these huge volumes of data to identify significant knowledge is not a trivial task. This task is especially difficult if we do not know that particular information or knowledge exists or can be deduced from the database. Thus, it is evident that intelligent automated tools are needed to analyze, profit, and extract useful information from these large databases.

A definition of KDD is “The KDD process is the process of using data mining methods (algorithms) to extract (identify) what is deemed knowledge according to the specifications of measures and thresholds, using the database $F$ along with any required preprocessing, subsampling, and transformations of $F$.” $F$ is a set of facts [Fayy96a]. This definition implies that the KDD process involves many activities and many steps, including:

1. Understand the application domain. This step involves collecting all the relevant knowledge about the domain and defining the objectives of the analysis.
2. Create a target data set. Determine the set of variables, partition of the data, or sample on which the discovery process will be done.
3. Perform data cleaning and preprocessing. Elimination of noise and outliers, as well as the definition of strategies to handle missing values, noise, time sequence information, and normalization.
4. Perform data reduction and projection. Find the appropriate form to represent the data or variables. Use transformations to reduce the number of variables, or to project them to spaces where the discovery process has a greater possibility for success.
(5) Choose the data mining task. Decide the type of knowledge (patterns) to be mined. Classifications, regressions, clustering, sequences, and associations are examples of the patterns that can be mined.

(6) Select the data mining algorithms. Choose the method (algorithm) to extract the patterns selected in the previous step.

(7) Perform data mining. Search for the desired patterns using the selected algorithms. This step involves selecting the parameters to run the algorithms.

(8) Interpret the mined results and iterate over steps 1 to 7 to improve the mined results.

(9) Consolidate the results into discovered knowledge and resolve possible conflicts between known knowledge and the mined knowledge.

This particular sequence of steps does not imply a sequential process but rather an iterative one. Indeed, the KDD process is open to multiple loops (iterations) among these steps [Fayy96a].

As these steps show, data mining is one part of the overall process of knowledge discovery in databases. In this context, data mining is defined as [Fayy96a]: "a step in the KDD process consisting of particular data mining algorithms that, under some acceptable computational efficiency limitations, produces a particular enumeration of patterns $E_j$ over $F$." A pattern "is an expression $E$ in a language $L$ describing facts in a subset $F_E$ of $F$. $E$ is called a pattern if it is simpler than the enumeration of all facts in $F_E$." For example, a pattern could be "90% of the customers that buy product A in a visit, buy product B in the next visit."

In summary, data mining is the search for patterns in a data set. The search has to be done within certain acceptable performance limits. This definition does not restrict the type or form of the patterns that can be mined, suggesting that data mining can produce different types of "knowledge." In general, the two primary goals of data mining are prediction and description.
Prediction focuses on using the values stored in the database to predict missing or future values. Description aims to find expressions that describe the data.

These data mining objectives can be modeled using the following most common model functions [Fayy96c]:

- **Classification.** The objective is to classify a data item into one of several predefined classes. For example, data mining classification algorithms can be used to identify particular objects in huge image databases.

- **Clustering.** The idea is to group data items to form classes or clusters of data items according to some similarity function. In this case, the data mining algorithm defines the classes as opposed to classification where the classes are predefined. For instance, data mining clustering algorithms can be used to identify groups of homogeneous people to help develop a marketing plan.

- **Regression.** The objective is to map a data item to a prediction variable to predict the value of a certain set of attributes.

- **Summarization.** The aim is to find a compact description of the data set. An example of this function is the derivation of summary rules.

- **Dependency modeling.** The objective is to find significant dependencies among data items. The dependency can be expressed in structural terms or in quantitative terms. The former describes a dependency network, and the latter describes the strength of the dependency using a particular scale.

- **Link analysis.** The objective is to find relationships among the data items. For example, a link analysis may produce association rules. The objective is to find rules of the form “c% of the customers that buy product A also buy products B and D.” This kind of information can be used to design the floor plan of the store, the marketing strategy, or even to forecast inventory levels.
• Sequence analysis. The objective is to model patterns that occur over time. The idea is to model the required states that produce a particular sequence of events. One example of sequence analysis is the identification of sequences of events such as "if event A occurs, then c% of the time events B and D occur within the next t units of time." This information can be used to forecast equipment failures and stock booms.

• Change and deviation detection. The objective is to detect significant changes in data from a previously time-stamped state of the data.

Several techniques can be used to implement these data mining functions, including decision trees and rules, neural networks, nearest-neighbor classification algorithms, case-based reasoning, and Bayesian networks. Thus, there are different patterns that can be mined and different techniques to mine them. In addition, there are many algorithms to mine the same pattern. These algorithms differ in their performance and in the technique they use.

Consequently, there are many options to choose from to apply data mining to reverse engineering. We chose a data mining model function that has a "direct application" to reverse engineering. A direct application means that the pattern produced by the selected data mining function provides information that can be used to produce a reverse engineer artifact such as a design. One of the problems in recovering a design is to identify the relationships among the system components to create a model of the system. In contrast, the link analysis model function finds relationships among data items. Thus, link analysis is the data mining function used in this research. The rationale for this decision is that if a data mining technique can unveil unknown relationships among system components, it is possible to use this information to build a model of the architecture of the system. Another important factor for selecting link analysis as the data mining function in this research is the relative maturity of this specific area. Link analysis in general and mining association rules in particular is a well-defined problem with a well-defined and tested solution.
The objective of link analysis is to identify "multi-field correlations satisfying support and confidence thresholds" [Fayy96c]. A particular product of a link analysis is the derivation of association rules. Agrawal, Imielinski, and Swami [Agra93] introduced the problem of mining association rules from large databases of transactions. Specifically, they focused on basket analysis. Their original idea was to find associations among the items a customer buys. For instance, having a large database of transactions, each transaction containing all the products purchased by a customer in a particular visit, the goal is to produce rules of the form "90% of the times a customer buys milk, he or she also buys bananas."

Formally, the problem of mining association rules is defined as follows in [Agra93]: Let \( I = \{i_1, i_2, i_3, \ldots, i_m\} \) be a set of items. \( D \) is a set of transactions \( R \) such that \( R \subseteq I \). Additionally, \( R \) contains \( X \) if \( X \subseteq R \). An association rule is an implication \( X \Rightarrow Y \), where \( X \subseteq I \), \( Y \subseteq I \), and \( X \cap Y = \emptyset \). The rule \( X \Rightarrow Y \) holds in \( D \) with confidence \( c \) if \( c\% \) of transactions in \( D \) that contain \( X \) also contain \( Y \). In addition, the rule \( X \Rightarrow Y \) has support \( s \) if \( s\% \) of the transactions in \( D \) contain \( X \cup Y \). Then, the problem of finding association rules in a set of transactions consists on finding all the association rules having \( s > \text{minsup} \) and \( c > \text{minconf} \). Minsup and minconf are user-supplied parameters representing the minimum required support and confidence, respectively.

For example, assume that a small supermarket sells salt, milk, soap, rice, and gum. Assume also that six customers have purchased products in this supermarket. Customer one bought salt and soap; customer two milk and rice; customer three milk, soap, rice, and gum; customer four soup, rice, and gum; customer five salt, milk and rice; and customer six salt, soup, and rice. The sets \( I \) and \( D \) are given by \( I = \{\text{salt, milk, soap, rice, gum}\} \), \( D = \{\{\text{salt, soap}\}, \{\text{milk, rice}\}, \{\text{milk, soap, rice, gum}\}, \{\text{soup, rice, gum}\}, \{\text{salt, milk, rice}\}, \{\text{salt, soup, rice}\}\} \) respectively. Table 1 presents a table view of the information.
Table 1. Example of an input set to mine association rules

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Products</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>milk</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
</tr>
<tr>
<td>6</td>
<td>X</td>
</tr>
</tbody>
</table>

Milk and rice appear together in three transactions. Two association rules can be derived from this observation. The first association rule is “milk ⇒ rice with a confidence of 100% and support 50%.” The confidence of the rule is given by the fact that three out of three transactions that contain milk also contain rice (i.e., 3/3 or 100%). The support is 50% because 3 out of six transactions contain the rule. The second association is “rice ⇒ milk with a confidence of 60% and support 50%.” The confidence is computed by taking the number of transactions that contain the rule divided by the number of transactions containing the left side of the rule (i.e., 3/5 or 60%). The support is the same as in the previous rule.

Obviously, the first rule is strongest with a confidence of 100%. The first rule means that every time a customer buys milk he or she also buys rice, while the second rule means that only 60% of the times a customer buys rice he or she also buys milk. Thus, the larger the confidence of an association rule the strongest the certainty of the rule. The support of an association rule gives a measure of the frequency of the rule in the transaction set. An association rule with large support implies that it is contained in most of the transactions while an association rule with small support is present in few transactions.

This example shows that mining association rules is a two step process. In the first step, the sets of items that satisfy the minconf and minsup parameters are found. The set \{milk, rice\} satisfies values of 50% for confidence and support. In the second step, the association rules are built. The set \{milk, rice\} produces the two association rules described above.
The following example shows an association rule that contains sets in either the left or right hand side instead of single items (products) (i.e., given the rule X \Rightarrow Y, X or Y is a set). Assume that the set \{soup, rice, gum\} satisfies the required confidence and support threshold values. Thus, it is possible to define the following association rule “gum \Rightarrow \{soup, rice\}” with 100% confidence and 33% support” which means that every time a customer buys gum he or she buys soup and rice.

The basic meaning of an association rule is that the transactions in a database that contain a data item A tend to contain a data item B. This information can be used to design catalogs, to define the layout of the store, to define marketing strategies, to forecast inventories, and for customer segmentation. The application domain of association rules has expanded outside basket analysis. Association rules have been applied in diverse domains ranging from decision support to diagnosis and prediction of alarms in telecommunication systems [Fayy96a].

Chen points out that the process of mining association rules might need to traverse the database several times, which leads to high processing times [Chen96]. Therefore, performance improvement is a major concern when designing algorithms to mine association rules. Several such algorithms have been proposed. First, the Apriori and AprioriTid algorithms [Agra94] improved the performance of the original algorithm presented in [Agra93]. Later algorithms are described in [Mann94], [Park95], and [Sava95]. In addition, some variants of the problem are addressed by algorithms to mine generalized association rules [Srik95], to mine multiple-level association rules [Han95], to mine association rules in distributed databases [Cheu96], and to mine quantitative association rules [Srik96].

2.2. Related Research

This section presents a review of literature in the reverse engineering domain. The first section focuses on reviewing research works on design recovery because design recovery is the general objective of the methodology defined in this research. The section also includes a
review of related research in subsystem classification, which is the specific goal of the methodology. Finally, this section includes a discussion of techniques and approaches in reverse engineering that use similar ideas to those used in mining of association rules.

2.2.1 Design recovery

According to the taxonomy of reverse engineering in [Chik90], design recovery is “a subset of reverse engineering in which domain knowledge, external information, and deduction or fuzzy reasoning are added to the observations of the subject system to identify meaningful higher level abstractions beyond those obtained directly by examining the system itself.” Design recovery focuses on producing abstractions of the subject system to facilitate the understanding of it.

There are different approaches to design recovery, and these approaches produce different results (e.g., different types of abstractions). Design recovery techniques can use any available source of information such as source code, documentation, domain knowledge, and personal experience. Similarly, the identified abstractions may take different forms such as module breakdown, structure-charts, entity-relationship diagrams, and formal specifications.

Normally, the source code is the preferred source of information of design recovery techniques. However, some research works explore the possibilities of using non-code sources of information. For example, Leite and Cerqueira [Leit95] have worked on recovering high level abstractions from structured specifications. They describe the process to extract business rules and some domain information from structured analysis specifications. They argue that their technique can be partially automated. Butler [Butl95] provides a semantic model to understand formalized data flow diagrams, arguing that a semantic model is indispensable for extracting useful information from a software document.

Formal approaches use rigorous mathematical procedures and notations to extract a formal description of the subject system. The advantage of such a description is that it is
precise, verifiable, and prone to automation. The REDO project (REngineering, DOnumenting, and validation of systems) at Oxford University focuses on understanding programs through formal methods. Specifically, REDO aims to extract formal specifications from COBOL programs using transformations from formalism to formalism [Bowe93] [Lano93]. Cimitile [Cimi95] uses symbolic execution to extract functional specifications from modules. He uses first order logic formulas called preconditions and postconditions to express the extracted specifications. Gannod and Cheng [Gann95] use the strongest postcondition predicate transformer as the formal basis to reverse engineer imperative code.

Knowledge-based works include some sort of knowledge in the design recovery process. Biggerstaff [Bigg89] indicates that an automated design recovery system would need a knowledge base or domain model that captures the expertise of a system expert. To apply his ideas, Biggerstaff provides a conceptual design of a design recovery system, called DESIRE, that incorporates source code, program documentation, a domain model, design knowledge, a reuse library, and a design recovery facility. In another work, Harris [Harr95] describes a framework to recover architectural design information where a library of architectural styles and style components to identify architectural elements is used.

There are many works that aim to save legacy systems using objects, that is, several approaches of design recovery focus on extracting an object-oriented representation of the subject system (a non-object-oriented source code). For example, the works of Jacobson [Jaco91] and Dietrich [Diet89] discuss the issues, strategies, and experiences in migrating non-object systems to object-systems. Furthermore, Newcomb and Kotik [Newc95] claim that they have developed a tool that automatically transforms a “system composed of procedural programs into a functionally comparable object-oriented system.” Their tool works with COBOL systems and is based on mapping COBOL syntactic units to object classes. For example, they define a data object class whose instantiations are records (i.e., paragraphs.
labeled with 01), a program object class whose instances are programs, and a procedure object class whose instances are paragraphs.

There has been an increased interest on object recovery or the identification of objects within procedural programs. Gall and Klösch [Gall95] use a hybrid approach to object identification that integrates information extracted from source code, human expertise, external domain knowledge, and application-specific knowledge. They start by generating low-level design documents such as structure charts and dataflow diagrams (DFD). The DFD is the basic element for their object identification process. Another work by Sneed and Nyáry [Snee95] describes an approach to extract an object-oriented design from mainframe COBOL programs. Their approach identifies objects from the different software components of a mainframe program (i.e. maps, databases, job control procedures, and programs). They store information from these components in a relational database, which is later used to generate the objects. Moreover, they describe a tool that they have built called OBJECT-REDOC. This tool generates object-oriented documentation from COBOL programs in the form of object-attribute trees. In a different direction, Yeh [Yeh95] describes automatic and semi-automatic techniques to recover implicit abstract data types (ADTs) and object instances from procedural languages. The work includes an interactive tool, called OBAD, to recover ADTs and object instances. OBAD uses a graph representation of the abstract syntax tree of the code to identify candidate ADTs.

Rich and Wills [Rich90] focus on automatic identification of clichés. A cliché is a program structure or algorithm that programmers use constantly such as binary searches and hash tables. They claim that by recognizing clichés it is possible to reconstruct the design and to generate documentation automatically for a program. Woods, Quilici, and Yang [Wood98] formulate the design recovery problem as a constraint satisfaction problem. In particular, they
extract design concepts from code using a constraint-based approach. They claim that their approach is scalable and tractable.

In the area of structured design, Edwards and Munro [Edwa93] obtain a SSADM (Structured Systems Analysis and Design Method) representation from the source code of a COBOL system. They developed the RECAST (Reverse Engineering into CASe Technology) method to attain this objective. They base their approach on transformations to produce a system specification that contains data, process, and on-line specifications. They claim that the produced SSADM specification is a non-loss representation of the original system's design.

Most of the research on design recovery found in the literature aims to recover the design of programs. However, the recovery of architectural information of both programs and whole systems has become a common target of research. The next section presents a review of research in this area.

2.2.2 Subsystem classification

Architectural recovery is a design recovery subproblem. The architecture of a software system defines the overall organization of its components (e.g., procedures, types, files, and global variables) into subsystems and describes the relationships among the subsystems [Rumb91] [Lakh97]. The main problem when recovering the architecture of a software system is to organize its components into subsystems [Lakh97]. This problem receives different names in the literature such as subsystem classification, subsystem identification, modularization, system partitioning, system structure analysis, and program structure understanding. In addition, the problem of identifying objects in procedural languages may be seen as a subsystem classification problem in the sense that finding objects implies finding sets of system components.

Lakhotia [Lakh97] considers that the problem of subsystem classification (SC) is essentially a graph partitioning problem. Many subsystem classification research works use a
graph to represent the target system and graph-partitioning algorithms or other related techniques to divide the graph into several subgraphs (i.e., subsystems). For example, Choi and Scacchi [Choi90] describe a technique that uses graphs for subsystem classification. They produce a hierarchical design description of a software system using a module interconnection language (MIL). Their approach starts with analyzing the source code to generate its corresponding design description in MIL. This design description includes all the modules of the system along with the relationships among modules. The resources that the modules exchange among them define their relationships. They describe two ways to recognize a module. One way is to consider each function in a source-code file to be a module. The other way is to use the source-code file as a module. They opt for the latter option in their approach. Next, the hierarchical description of the system is generated using a graph-based restructuring algorithm. This algorithm uses a graph to represent the modules (nodes) and their relationships (edges). The algorithm is based on identifying articulation points. Therefore, if the graph has no articulation points, this approach cannot produce subgraphs (i.e., subsystems). They use a tree to represent the recovered subsystem classification in which each parent node is represented by a circle and denotes a subsystem, and each leaf node is represented by a square and denotes a module.

Lakhotia [Lakh97] defines a framework to express and classify subsystem classification techniques. He defines two classification criteria. The first criterion is the type of structure produced by the SC technique. The resulting structure can be either stratified or nonstratified. The former refers to tree-like structures where internal nodes represent subsystems and leaf nodes system components. The later refers to flat structures where the system is separated into several subsets of system components. The second criterion classifies the SC techniques according to the type of algorithms they use. Lakhotia proposes the following four categories. The SC technique is numeric if it uses variables that are defined by a ratio, an interval scale, or
by numeric computations. The technique is graph-theoretic if the SC technique uses graph-
theoretic algorithms. The technique is flow-analysis based if it is based on flow-analysis
concepts (flow-analysis is used for optimization in compilers). Last, the technique is
conceptual if the technique uses variables that are measured in an ordinate scale.

Lakhotia uses this framework to classify several subsystem classification techniques. He
classifies as numeric stratified the techniques proposed by Hutchens and Basili [Hutc85],
Maarek, Berry, and Kaiser [Maar91], Maarek and Kaiser [Maar88], Schwanke [Schw91], and
Selby and Basili [Selb91]. Lakhotia considers as numeric nonstratified the techniques
described by Belady and Evangelisti [Bela81], Achee and Carver [Ache94], and Patel, Chu,
and Baxter [Pate92]. He places the work of Choi and Scacchi [Choi90] in the category of
graph-theoretic stratified. In the graph-theoretic nonstratified class, he includes the techniques
proposed by Livadas and Johnson [Liva94], and Ogando, Yau, Liu, and Wilde [Ogan94]. The
works by Ong and Tsai [Ong93], and Silva-Lepe [Silv93] are regarded as flow-analysis based
and nonstratified. Finally, Lakhotia considers the work of Müller and Uhl [Müll90] as a mixed
stratified SC technique because it uses numeric computations as well as graph-theoretic
techniques. In this context, the ISA methodology proposed in this research can be classified as
numeric and stratified.

In the area of object identification as a SC technique, Canfora, Cimitile and Munro
[Canf96] describe an algorithm that uses a combination of a graph and statistical techniques to
identify objects in imperative code. Their work is similar to subsystem classification in the
sense that they define an object as a set composed of data items and routines (i.e., a
subsystem). Although they work at program level, their overall approach to the problem is
similar to the overall approach used in the ISA methodology. For example, their algorithm
represents the input and output graphs as tables, where the columns are the routines and the
rows are the data items. ISA uses also a table as the input to the mining process. Their

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
approach consists on representing the subject program as bipartite graph and then applying an iterative clustering algorithm to decompose the original graph into a set of unconnected graphs (i.e., objects). ISA uses a clustering algorithm to form sets of programs and files. The clustering part of their algorithm is based on statistical techniques to measure the internal connectivity of the subgraphs and to discriminate the graph components that would prevent object formation. The clustering part of ISA is guided by the mined associations and the internal connectivity of the sets is controlled by similarity functions. Thus, both techniques start with a table and incrementally build sets of system components based on some similarity measure.

In a different approach, Anquetil and Lethbridge [Anqu98] use the file names as the source of information to form clusters of concepts. They consider that these concepts may form subsystems. They based their work on the assumption that the designers and maintainers of large software systems follow some naming conventions for the files in the system. Thus, their work aims to classify the system’s file names into several concepts or subsystems. They do not specify if a file name refers to the name of the program source file, a data file, or both. Their classification technique starts by finding the abbreviations used to form the file names. They use several sources of information to extract candidate abbreviations such as file names, comments, identifiers, and the English dictionary. Next, they decompose the file names into a list of abbreviations. Finally, they use this information to map file names to concepts. However, they just provide an evaluation of the decomposition phase and do not provide any details on how to map file names to concepts (i.e., subsystem formation). Moreover, their technique does not suggest that it can produce a hierarchy of subsystems nor it provides a presentation model.
2.2.3 Other related research

The idea of using a database representation of the subject system is not new. Chen [Chen90] generates a relational view of C code to support software activities such as graphical views, subsystem extraction, binding analysis, dead code elimination, and program layering. Narat [Nara93] uses a database to support maintenance activities of source code by producing cross-reference documentation. Grass [Gras92] uses CIA++ (C++ Information Abstractor) to extract design information from C++ programs. CIA++ constructs a relational database that contains information obtained from C++ programs. Her aim is to do object recovery by querying the relational database created by CIA++. Although these works use a database representation of the subject system, data mining techniques are not used to extract design information.

Basic data mining techniques have been used in some research to support reverse engineering activities. Anthony Berglas and John Harrison [Berg97] report the use of basic data mining in the Ingres to Oracle Conversion (ITOC) reverse engineering tool. The ITOC tool “automatically recovers both the application structure and the static schema definition of Ingres 4GL applications, the results of which are then loaded into Oracle's CASE tool.” They query the application’s data to understand some structural issues of the subject system. For example, they use queries to determine if certain columns are mandatory or optional, to get specific ranges of column values, to determine if foreign keys belong to a one to one relationships, and to determine if numeric fields may be negative. They refer to these queries as “basic data mining.”

Another approach to subsystem identification is based on concept analysis. The underlying ideas behind concept analysis are similar to the ideas behind an association rule. Basically, concept analysis can be used to form groups of objects that have common attributes [Siff97]. To explain the underlying ideas of concept analysis, we use the example provided by
Siff and Reps in [Siff97]. The first step in concept analysis is to define a context. A context is a triple \( C = \{ O, A, R \} \) where \( O \) is a set of objects, \( A \) is a set of attributes, and \( R \) is a binary relation between \( O \) and \( A \). The example given by Siff and Reps uses the sets \( O = \{ \text{cats, chimpanzees, dogs, dolphins, humans, whales} \} \) and \( A = \{ \text{four-legged, hair-covered, intelligent, marine, thumbed} \} \). The following table shows the table definition of \( R \).

### Table 2. Example of a context

<table>
<thead>
<tr>
<th>Objects</th>
<th>four-legged</th>
<th>hair-covered</th>
<th>intelligent</th>
<th>Marine</th>
<th>thumbed</th>
</tr>
</thead>
<tbody>
<tr>
<td>cats</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>chimpanzees</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>dogs</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>dolphins</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>humans</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>whales</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A formal concept is a pair of sets \((O, A)\) such that \( O \subseteq O, A \subseteq A, A = \text{ca}(O), O = \text{co}(A) \). \( \text{ca}(O) \) denotes the set of attributes that are common to all objects in \( O \). Similarly, \( \text{co}(A) \) denotes the objects that are common to all attributes in \( A \). Thus, a formal concept is the largest set of objects with the same attributes. For instance, \((\{\text{dolphins, whales}\}, \{\text{intelligent, marine}\})\) is a concept, but \((\{\text{human, dolphin}\}, \{\text{intelligent}\})\) is not a concept.

Concept analysis has been applied to subsystem identification because it is possible to create a lattice of concepts. In other words, it is possible to arrange concepts in a hierarchy of concepts, subconcepts, sub-subconcepts, and so on. The concept lattice of the context in Table 2 is given in Figure 1. This figure shows all possible concepts from the context given in Table 2 hierarchically arranged.

This characteristic of concept analysis has been used for identification of modules in legacy code. Lindig and Sneltinig [Lind97] use this technique to modularize legacy code. They use the set of functions of a program as the set of objects, the set of global variables as the set...
of attributes, and the usage of global variables by the functions as the relation. They use this configuration to analyze Modula-2, Fortran, and COBOL programs. They were not able to produce a modularization of the programs due to the large amount of interconnected branches in the concept lattice.

Siff and Reps [Siff97] report better results. They use concept analysis to identify modules in C. They use functions as the set of objects and some function attributes as the set of attributes. Some examples of the attributes they use are type of the returning value of the function, type of the arguments, and type of the internal variables of the function. They report a successful decomposition of C programs. They consider that their success is due to the language chosen, and to the use of "negative information" (e.g., function X does not use variables of type Y). The main problem of this approach is the selection of the right attributes that lead to a decomposable concept lattice. Otherwise, this approach faces the same problem as Lindig and Snelting, namely too many interconnections in the concept lattice. Therefore, the selection of the attributes requires certain knowledge of the problem, the data types, and the global data structures.

Although these research works on module identification do not produce the same output as the methodology presented in this research, they are included in this review because of their

![Figure 1. Example of a concept lattice](image)

Concepts:

- **Top**: \(\{\{\text{cats, chimpanzees, dogs, dolphins, humans, whales}\}, \{\}\}\)
- **C5**: \(\{\{\text{chimpanzees, dolphins, humans, whales}\}, \{\text{intelligent}\}\}\)
- **C4**: \(\{\{\text{cats, chimpanzees, dogs}\}, \{\text{hair-covered}\}\}\)
- **C3**: \(\{\{\text{chimpanzees, humans}\}, \{\text{intelligent, thumbed}\}\}\)
- **C2**: \(\{\{\text{dolphins, whales}\}, \{\text{intelligent, marine}\}\}\)
- **C1**: \(\{\{\text{chimpanzees}\}, \{\text{hair-covered, intelligent, thumbed}\}\}\)
- **C0**: \(\{\{\text{cats, dogs}\}, \{\text{four-legged, hair-covered}\}\}\)

**Bottom**: \(\{\}, \{\text{four-legged, hair-covered, intelligent, marine, thumbed}\}\)
approach. These research works use concept analysis, which has some similarities to association rules. For instance, concept analysis and mining association rules use a matrix as input. In concept analysis, the rows represent objects and the columns attributes. In mining association rules, the rows represent transactions and the columns represent items. Both techniques aim to detect sets of rows that intersect a similar set of columns. However, in concept analysis this grouping is maximal and unique. In mining association rules, the grouping is more flexible because it is determined by the values defined for the confidence and support of the rules. Moreover, an association rule can contain any number of objects related to any number of attributes. This flexibility allows the definition of algorithms to form groups of objects and attributes (itemsets) incrementally as opposed to concept analysis where the grouping is fixed. Moreover, it is possible to build a concept lattice or a similar construction based on association rules. The difference is that with an algorithm based on association rules it is possible to avoid the branches in the lattice that can lead to a non-decomposable lattice.

2.3. Design Recovery and Data Mining

In sections 2.1 and 2.2, we described the main theories supporting this research. In this section, we introduce the underlying ideas of this research. Section 2.3.1 presents the motivation to pursue this data mining approach to reverse engineering. Section 2.3.2 introduces a general method to apply data mining techniques in the reverse engineering and maintenance domains. This general method serves as a template to develop specific methodologies to apply data mining to design recovery.

2.3.1 Motivation

Any attempt to reverse engineer, maintain, or enhance large software systems includes some of the following problems: unstructured programming and design, outdated documentation, undocumented patches, lack of expert availability, lack of naming standards,
and use of cryptic component names. Furthermore, the size of these systems increases the complexity of the task. Reverse engineers and maintainers require techniques capable of extracting useful information from systems that are large, complex, poorly documented, and poorly understood. Data mining has some features that suggest that it can be used to address some of these problems. For instance, while reverse engineering has to cope with large legacy systems, data mining is adept at dealing with a great deal of information.

The following data mining features suggest the possible use of data mining in reverse engineering and maintenance.

(a) Data mining can discover unsuspected non-trivial patterns and relationships among data elements in large databases. This feature can be used to address the problem of eliciting the relationships among system components. The subject systems of a reverse engineering analysis have gone through continuous maintenance, thereby many changes. These changes produce a loss of structure of the system that is reflected in the lack of clear relationships among its components. In addition, patches and enhancements to the systems lead to more intertwined relationships among their components. These relationships produce a complex network of component interdependencies. Data mining could be used to identify and unwind these complex relationships among system components. Furthermore, data mining could elicit new knowledge about the subject system in the form of relationships and patterns. This information could be used to support reverse engineering and maintenance tasks such as design recovery, object extraction, identification of reusable parts, and detection of repeated code.

(b) Data mining techniques are capable of eliciting relevant information without any previous knowledge of the object of study. This data mining feature is very useful in reverse engineering and maintenance, especially when dealing with legacy systems. As
mention before, some of the common problems that a reverse engineer faces are the lack of proper documentation, lack of naming standards, cryptic names, and poor system and program design. In addition, it is common that the original developers and maintainers are not available. That is, there is lack of expert availability. The reverse engineer or maintainer has to make the analysis of the system with little reliable information. Often, the source code is the only piece of such an information; however, identifying significant information from hundreds of thousands of lines of code is not a simple task. Data mining has the potential to produce relevant system information without any previous knowledge of the system's functionality and implementation details.

(c) Data mining is designed to analyze large databases. A major challenge for reverse engineering and maintenance is the analysis of large systems. The larger the system the more complex the analysis. Tools and techniques that work well in small systems may not work in large ones. In addition, the analysis of large systems requires automatic tools. A technique that cannot be automated is of little use. These problems can be overcome by using data mining. Data mining is capable of eliciting information in large volumes of data, it is scalable, and it is automated. Thus, data mining has the capability to deal with the complexity of the analysis of large software systems. The larger the system the better chances that data mining will produce significant information.

These data mining features position data mining as a sound concept for reverse engineering. In particular, these observations suggest that major problems in reverse engineering can be addressed by data mining techniques providing that these problems can be redefined in terms of knowledge discovery problems. Thus, the main motivation for this research is to explore and define the specifics on how to align data mining and reverse engineering.
2.3.2 The three-step method

The next step is to define the "how." This research implements the "how" in three levels. It defines a general method, a methodology, and a tool. Ghezzi, Jazayeri, and Mandrioli [Ghez91] establish a relationship among the terms method, methodology, and tool. They consider that these terms relate hierarchically to each other in the following way. At the bottom, the methods define the general approach to solve a problem. At the next level, the methodologies define the specific steps to solve the problem. These steps are based on the guidelines given by the selected method. At the top of the hierarchy, the tools support the application of the methodology to solve a particular problem.

This taxonomy served as the framework for the development of the main components of this research. In this research, we define a method, called the three-step method (TS); we define a methodology, called the ISA methodology; and we use a tool, named RE-ISA. This section describes the TS method to apply data mining to reverse engineering and maintenance. Chapter 3 describes the ISA methodology in detail, and Chapter 5 describes the use of the RE-ISA tool.

In order to use data mining techniques for reverse engineering it is necessary to express the reverse engineering problem as a knowledge discovery problem. Then, it is necessary to combine the discovered patterns into a useful high-level abstraction. This observation led to the definition of a general method to apply data mining to reverse engineering. The idea is to use this general method as a framework to define particular methodologies that use specific data mining techniques and that produce specific reverse engineered artifacts.

This general method consists of three steps:

1. Define a database view of the system. A database view of the system is a representation of the system or a subset of it using a database. The data to be loaded into this database come primarily from the source code (e.g., variables, programs, modules,
and files), but it may come from any other source of information. The selection of the database view determines the type of information that the data mining algorithms can mine. Consequently, the selection of this view is critical to the success of the mining analysis, and it is done with the selection of the particular data mining algorithm in mind.

2) Perform data mining. This step involves the selection and use of data mining algorithms to mine the database view of the system. The selection of data mining algorithms depends on the specific information requirements of the reverse engineering process. For example, the database view may be mined to search for associations, sequences, classifications, or clusters.

3) Consolidate and interpret results. The outcome of the mining process is combined into meaningful knowledge to construct the desired reverse engineering artifact (e.g., a design of the system, documentation, and subsystem classification). Although the desired reverse engineering artifact is constructed in this step, it is necessary to define this artifact first. The target artifact determines the selection of the database view and the data mining algorithms.

This three-step method can be seen as a customization of the nine-step KDD process. However, this customization works backwards in the sense that the desired recovery artifact is defined first. Then, the database is designed specifically for the data mining task. The selection of the required knowledge precedes the selection of the right database view and the right data mining algorithms. In contrast, in a regular KDD process the database is already defined and it may contain noise, missing values, and outliers. As in the KDD process, this three-step method does not imply a linear sequence of steps; rather it is open to iterations. For example, steps one and two may be applied several times before going to step three. This
means that the generation of the reverse engineering artifact may be based on several data mining algorithms mining knowledge from different database views of the target system.

The TS method (three-step method) is a general framework because it can be used to define specific methodologies to extract specific reverse engineering artifacts. Moreover, the TS method can be applied at different granularity levels. For instance, it has potential to be used to analyze a program, a module, or a system. It can produce diverse high level abstractions depending on the specific instantiation of the database view, the mining algorithms, and the consolidation procedure.

Following this definition of a general method to use data mining in reverse engineering, the next step is to define an instantiation of this method. Such an instantiation provides evidence that the three-step method works and that data mining is a valuable tool for reverse engineering and maintenance. Chapter 3 describes an instantiation of the TS method, called the ISA methodology. Chapter 4 describes a graphical model to present the outcome of ISA. Then, chapter 5 describes experiments using ISA, and results providing positive evidence of the feasibility and potential of this data mining approach to reverse engineering.
CHAPTER 3. THE ISA METHODOLOGY

In this chapter, we describe the ISA (Identification of Subsystems based on Associations) methodology. Section 3.1 provides a summary of the development of the ISA methodology. It includes the motivation, objectives, and rationale of the design of ISA. In section 3.2, we present a general description of the ISA methodology. This section includes the description of the input, the output and the main steps of ISA. Finally, sections 3.2 to 3.5 describe each of the three major steps of the ISA methodology.

3.1. Introduction

Before describing the ISA methodology, it is important to explain the motivation and objectives that lead to its definition. This contextualization of ISA helps to understand the design decisions and the general approach used to define it. As stated before, the general objective of this research is to show that data mining is a valuable tool for software reverse engineering and software maintenance. The approach used to demonstrate the value of data mining in the reverse engineering and maintenance domains consists of the following steps. First, we devised a general method to apply data mining to reverse engineering problems (i.e., the TS method presented is section 2.3.2). Then, we used the method to develop a specific methodology for a reverse engineering problem (i.e., the ISA methodology presented in this chapter). Finally, we evaluated the methodology on real-world software systems (the case studies are presented in chapter five).

Several requirements guided the definition of ISA. First, ISA should be based in the three-step method. Second, ISA should be able to analyze large software systems. Third, ISA should produce a high-level abstraction that can be used for reverse engineering or maintenance of software systems. Lastly, ISA should be amenable to automation.

The first requirement provided the framework to design the methodology. The three-step method defines a database view of the subject system, mines this database view, and finally
consolidates the mined knowledge into a meaningful reverse engineering artifact. Therefore, the initial design decisions were the selection of the database view, the selection of the data mining technique, and the selection of the reverse engineering artifact to be constructed.

As explained in section 2.1, we selected mining association rules as the data mining technique for this research. Since association rules elicit relationships among the elements in a database, the next decision was to select the system components to be stored in the database. In other words, it was necessary to select the system components whose relationships could be used to create a reverse engineering artifact. We selected programs (i.e., source files) and data files as the components. The rationale for this decision is that by identifying relationships between programs and data files it could be possible to cluster the programs and data files with the strongest relationships. These clusters would represent subsystems. Hence, the initial design decisions were the use of programs and data files to create the database view of the subject system, the mining of association rules, and the derivation of a subsystem decomposition of the subject system.

The definition of the structure of the specific database view was straightforward because the problem of mining association rules has a well-defined input, a table. The issue was to decide how to form the table. It was necessary to decide whether the rows represent programs or data files. If rows represent programs, then the columns would represent data files. However, if rows represent data files, then the columns would represent programs. Moreover, it was necessary to define the meaning of the intersection of a column and a row. The alternatives considered were that the intersection of a column and a row would be marked if

(a) The program reads the file.
(b) The program writes to the file.
(c) The program updates the file.
(d) The program uses the file in any way (i.e., reads, writes, or both).
After several experiments, we decided to mine two tables. In one table, the rows represent programs, and in the other, the rows represent data files. In both cases, the intersection of a row and a column is marked if the program (represented by the row or column) uses the data file (represented by the column or row) in any way.

Although the original objective was to decompose a system into several subsystems, the nested structure of association rules led to the generation of a hierarchical subsystem decomposition instead of the flat decomposition considered initially. Moreover, this characteristic of association rules also produced an unexpected relationship among the data files in the system. That is, certain data files can be arranged in a hierarchy of file implications. Hierarchies of file implications are explained in section 3.2. These outcomes were unexpected added-value results produced by the specific data mining technique used (i.e., the natural nested structure of association rules).

A final word on the development of ISA. The first approach was to build a table that represents the programs in its rows and the files in its columns. Then, mined associations are used to rearrange the rows and the columns in the table in such a way that the programs forming a subsystem are in adjacent rows. Similarly, the files that tend to be used together are in adjacent columns. This approach to subsystem identification was abandoned because there was not evidence that this process was automatable. A description of this approach is presented in [Mont98].

3.2. Overview

ISA is a system level methodology that decomposes a software system into a hierarchy of data cohesive subsystems. This definition contains three basic concepts. First, ISA is a system level methodology. Second, ISA decomposes a system into a hierarchy of subsystems. Last, the subsystems that ISA identifies are data cohesive.
ISA is a system level methodology because its object of analysis is a complete software system as opposed to other subsystem classification techniques whose object of analysis is a program or a module. Therefore, the input to the ISA methodology is a software system. Specifically, ISA accepts a software system $S$ composed of a set of programs $\mathcal{P}$ (i.e., source files) and a set of data files $\mathcal{F}$. A typical example of such a system is a human resources system written in a third generation language such as COBOL. This system would likely be composed of several subsystems such as payroll, training, recruiting, and benefits. Each subsystem would include several programs and several data files. For instance, the payroll subsystem may include programs to print the payroll, to print checks, to perform the calculations, and to report tax withheld. In addition, it would contain several data files such as the roster file, the salaries file, and the scheduling file. The system may also include data files that are used by several subsystems such as the master employee file and the organizational units file. For simplicity, this definition of this system does not include script files and JCL (Job Control Language) scripts. In addition, we use the terms "data file" and "file" interchangeably.

The output of the ISA methodology is a decomposition of $S$ into a hierarchy of disjoint data cohesive subsystems. In general, ISA defines a subsystem as a set $Z=\{G, H\}$ such that $G \subseteq \mathcal{P}$ and $H \subseteq \mathcal{F}$. Hence, ISA decomposes $S$ into $k$ subsystems $Z_i=\{G_i, H_i\}$ for $i=1,2,\ldots,k$, where $G_i \cap G_j = \emptyset$, and $H_i \cap H_j = \emptyset$ for $i, j = 1, 2, \ldots, k$, and $i \neq j$. However, $G_1 \cup G_2 \ldots \cup G_k$ may not be equal to $\mathcal{P}$, and $H_1 \cup H_2 \ldots \cup H_k$ may not be equal to $\mathcal{F}$ because there are some programs that cannot be classified into any subsystem and some files that are used by several subsystems (i.e. the master employee file in the example above).

ISA organizes the $k$ subsystems in a hierarchy of subsystems. ISA joins the identified subsystems to form larger subsystems (i.e., suprasystems containing subsystems). These larger
subsystems are merged to form even larger subsystems. This process continues until the largest subsystems reach a dissimilarity threshold.

In addition, the subsystems that ISA generates have the characteristic that the programs $G_i$ in a particular subsystem $Z_i=\{G_i, H_i\}$ access primarily the files in $H_i$. In other words, the programs in a subsystem use predominantly the files in the subsystem. However, this subsystem decomposition does not imply that a program in a subsystem $Z_i$ cannot use a file in a subsystem $Z_j$. Rather, it means that the files in a subsystem are used predominantly by the programs in the same subsystem. That is, the programs in a subsystem access the same data repositories. In that sense, ISA produces data cohesive subsystems.

From this general description, it can be deduced that ISA uses static analysis. That is, ISA relies on analyzing the source code to produce the subsystem decomposition. No runtime analysis is performed.

3.2.1 The input

The source of information that ISA uses is the source code. Therefore, the input of the ISA methodology is the set of files that contain the source code that produced the current running version of the system to be analyzed. As other analysis tools, the ISA methodology assumes that the source code is updated. This is a fair assumption since the source code normally corresponds to the running code. In cases where the source code is lost or is outdated, the ISA methodology cannot be used.

The generic system that ISA can decompose into subsystems is a system composed of several identifiable portions of code and several independent data repositories. The portions of code are called programs and the data repositories files. In addition, there should be a clear way to identify which files are used by each program. In this research, a program uses a file if it reads or writes information on it.
An example of a system that ISA can analyze is a COBOL system composed of several source files (i.e., programs) and several flat data files. However, ISA is capable of analyzing systems written in other languages and systems that use other types of persistent data management (e.g., databases). For example, ISA can decompose a software system developed in C with embedded calls to a relational database. The programs can be defined as the source code files that have a "main" subroutine. The files would be the tables defined in the database.

Another example of a system that can be analyzed with ISA is a system written in a proprietary 4GL language such as Progress. In this case, the programs are the so-called modules and the files are the tables defined in the relational database.

The architecture of the system, the programming style, and the specific programming language affect what is called a program and what is called a file. For example, if the system stores all the data in a single data repository, then ISA is of little use. It would be necessary to identify sets of data elements that can be packed as a single unit (i.e., a file). Moreover, if the system is programmed as a monolithic large piece of code, then the criteria of calling each independent source code file a program does not make sense. In this case, another criteria should be used such as considering sections, modules, or even functions as the main source-code unit. Finally, certain programming languages may require special analysis to define a program. For example, COBOL, C, and other languages allow "include" files. Therefore, it is necessary to determine if these include files can be considered as a separate program or just a part of the calling program. Another example of this issue is the implementation of subprogram calls. For instance, COBOL programs are called through the Linkage Section. Thus, it is necessary to define a criterion to determine whether the code called through the Linkage Section is considered part of the caller or a separate program.
3.2.2 The output

The main outcome of the ISA methodology is a software system that has been decomposed into a hierarchy of subsystems. To form the hierarchy, ISA produces two types of subsystems: primitive subsystems and complex subsystems. Primitive subsystems contain programs and files, and complex subsystems contain primitive subsystems and files. Empirically, primitive subsystems correspond to leaf nodes and complex subsystems correspond to internal nodes in the tree representing the hierarchical subsystem decomposition. In other words, a primitive subsystem has a parent node and no children. A complex subsystem has a parent (except for the root node) and one or more children nodes.

Let \( S = \{ P, F \} \) be the input software system where \( P \) is the set of programs and \( F \) is the set of files. The set of primitive subsystems is denoted by \( Z \) and the set of complex subsystems is denoted by \( S \). A primitive subsystem is a set \( Z = \{ G, H \} \) such that \( G \subseteq P \) and \( G \neq \emptyset \), and \( H \subseteq F \) or \( H = \emptyset \). Thus, a primitive subsystem is composed by a set of programs and a possible empty set of files. A complex subsystem is a set \( S = \{ E, I, K \} \) such that \( E \subseteq S \) or \( E = \emptyset \), \( I \subseteq Z \) and \( I \neq \emptyset \), and \( K \subseteq F \) or \( K = \emptyset \). Hence, a complex subsystem contains \( n \) complex subsystems where \( n \geq 0 \), at least one primitive subsystem, and zero or some files.

The sets of programs are disjoint. Therefore, \( G_i \cap G_j = \emptyset \) for \( i \neq j \) and \( i, j = 1, 2, \ldots, |Z| \). Similarly, the sets of files are disjoint. That is, \( H_i \cap H_j = \emptyset \) (for \( i \neq j \) and \( i, j = 1, 2, \ldots, |Z| \)), \( K_i \cap K_j = \emptyset \) (for \( i \neq j \) and \( i, j = 1, 2, \ldots, |S| \)), and \( H_i \cap K_j = \emptyset \) (for \( i = 1, 2, \ldots, |Z| \) and \( j = 1, 2, \ldots, |S| \)). In other words, a program is assigned to only one primitive subsystem, and a file is assigned to only one subsystem that is either a primitive or a complex subsystem. This decomposition does not mean that programs in different subsystems cannot access the same file. However, the decomposition arranges the programs and files in subsystems to facilitate reverse engineer and maintenance activities.
There is a distinction between the files assigned to primitive subsystems and the files assigned to complex subsystems. A file assigned to a primitive subsystem \( Z_i \) is used primarily by the programs in \( Z_i \). A file assigned to a complex subsystem \( S_i \) is used primarily by programs in primitive subsystems that have \( S_i \) as an ancestor. Empirically, a complex subsystem is a set of primitive subsystems. Therefore, files assigned to a complex subsystem can be seen as shared files in the sense that programs in different primitive subsystems use these files.

ISA decomposes the system into one or more subsystem hierarchies. That is, the resulting decomposition may not be a single hierarchy tree but a forest. In this case, there are several complex subsystems with no parent subsystem (i.e., several roots). These subsystems are called main subsystems. One interpretation of this behavior is that the main subsystems constitute reasonable independent subsystems. If a single tree is desirable, ISA provides a mechanism to organize the main subsystems into a single hierarchy tree. This merger procedure creates a new type of supra subsystems in the sense that these new subsystems are composed exclusively of complex subsystems. Figure 2 shows a graphical representation of the type of subsystem decomposition that ISA produces.

ISA produces other interesting outcomes and byproducts such as unconnected programs, singular programs, unconnected files, common files, independent files, hierarchies of file implications, and link files. In the following paragraphs we describe these ISA byproducts.

First, \( G_1 \cup G_2 \cdots \cup G_P \) may not be equal to \( \mathcal{P} \). There are two components of this result. The first component derives from the fact that some programs use just one file or no files. These programs are not included in the mining process because a program has to use at least two files to produce an association. These programs are called unconnected programs. Unconnected programs are not assigned to any primitive subsystem because they do not share any data repositories with other programs. The second component refers to programs that
cannot be assigned to any primitive subsystem even though they use more than two files. These programs are called *singular programs*. Singular programs do not fit any primitive subsystem because the mining process does not find associations that involve these programs. This behavior suggests that singular programs may process special conditions such as exceptions or one-time reports.

![Graphical representation of a subsystem decomposition](image)

**Figure 2.** Graphical representation of a subsystem decomposition

Similarly, \( H_1 \cup H_2 \cup \ldots \cup H_{|H|} \cup K_1 \cup K_2 \cup \ldots \cup K_{|K|} \) may not be equal to \( \mathcal{S} \). As in the previous case, there are two factors for this behavior. First, some files are used just by one program. Thus, these files are not included in the database view and consequently are not assigned to any subsystem. These files are regarded as *unconnected files*. Unconnected files may be temporary files because there are used just by one program in the system. Second, some files cannot be assigned to any particular subsystem because they are used by programs from different subsystems. These files are called *common files*. An example of a common file
is the master employee file in a human resource system. The master employee file cannot be assigned to any particular subsystem (e.g., payroll, benefits, scheduling) because this file is used by all the subsystems. In other terms, a file is assigned to a particular subsystem $X$ when the majority of the programs that use this file are inside $X$. However, a common file is used by programs in many subsystems; thereby the file cannot be assigned to any particular subsystem. That is, there is no subsystem containing most of the programs that use the file.

Although common files are not assigned to any subsystem, they play an important role in the system. Common files can be seen as the major links among main subsystems. For example, they may serve as central repositories or as major communication buffers among subsystems.

Common files are used to define *hierarchies of file implications*. However, not all the common files are part of a hierarchy of file implications. If a common file does not belong to a hierarchy of file implications, it is said that the common file is an *independent file*. A hierarchy of file implications is a set of file implications that form a hierarchy. For example, assume that $a$, $b$, and $c$ are files, and that the mining process produced the following two association rules: $b \rightarrow a$ (100%), and $c \rightarrow b$ (100%). The former rule means that 100% of the programs that use file $b$ also use file $a$, and the latter rule means that 100% of the programs using file $c$ also use file $b$. Clearly, $b$ implies $a$ and $c$ implies $b$. Thus, $c$ implies $a$. In other words, if a program uses file $c$, then it also uses files $b$ and $a$. Hence, these files form the following hierarchy of file implications: $c \rightarrow b \rightarrow a$. Hierarchies of file implications are another interesting result derived from mining associations that represent novel architectural information.

Finally, some of the files assigned to a particular subsystem may be used by programs in other subsystems. These files are called *link files*. A link file is assigned to a particular subsystem $X$ because most of the programs that use it are in $X$. Nevertheless, this link file is
used by few programs outside $X$. Link files can be seen as the communication interface among subsystems. For each subsystem $X$ having files assigned to it, ISA identifies the set of subsystems and the number of programs in each of these subsystems that use files in $X$. Similarly, ISA identifies the files assigned to other subsystems that are used by the programs in $X$. Table 3 summarizes the information produced by ISA.

Table 3. ISA products

<table>
<thead>
<tr>
<th>Product</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive subsystem</td>
<td>Subsystem composed of a set of programs and $n$ files, where $n \geq 0$</td>
</tr>
<tr>
<td>Complex subsystem</td>
<td>Subsystem composed of at least one primitive subsystem, $n$ complex subsystems, and $m$ files, where $n \geq 0$ and $m \geq 0$</td>
</tr>
<tr>
<td>Main subsystem</td>
<td>Complex subsystem that is not an element of any other complex subsystem (i.e., complex subsystem with no parent)</td>
</tr>
<tr>
<td>Supra subsystem</td>
<td>Subsystem composed of complex subsystems</td>
</tr>
<tr>
<td>Unconnected programs</td>
<td>Programs that are not included in the mining process</td>
</tr>
<tr>
<td>Singular programs</td>
<td>Programs included in the mining process but not assigned to any primitive subsystem</td>
</tr>
<tr>
<td>Unconnected files</td>
<td>Files that are not included in the mining process</td>
</tr>
<tr>
<td>Common files</td>
<td>Files included in the mining process but not assigned to any subsystem</td>
</tr>
<tr>
<td>Independent files</td>
<td>Common files that do not belong to any hierarchy of file implications</td>
</tr>
<tr>
<td>Hierarchies of file implications</td>
<td>Hierarchical organization of association rules that relate files</td>
</tr>
<tr>
<td>Link files</td>
<td>Files assigned to a particular subsystem but used by programs outside that subsystem</td>
</tr>
</tbody>
</table>

3.2.3 The process

ISA consists of three major phases corresponding to each of the three steps of the TS method. Specifically, ISA instantiates the TS method as follows. The first step of the TS method calls for the definition of a database view of the system. ISA uses a set of tuples as the
database view of the system. The next step is to perform data mining over the database view. ISA mines the set of tuples in search for association rules. In the last step, the results of the data mining process are consolidated into a high-level abstraction. In this step, ISA uses four algorithms to produce the outcome described in the previous section.

These three major phases are subdivided in steps. Thus, the ISA methodology consists of 11 steps. The organization and sequence of the eleven steps are given in Figure 3. A description of each of these steps follows.

<table>
<thead>
<tr>
<th>I. Build a database view of the system.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Parse the source code</td>
</tr>
<tr>
<td>2. Assign unique identifications</td>
</tr>
<tr>
<td>3. Generate the alpha set</td>
</tr>
<tr>
<td>4. Build the alphaT and alphaN sets of tuples</td>
</tr>
<tr>
<td>II. Perform data mining</td>
</tr>
<tr>
<td>5. Mine the alphaT set</td>
</tr>
<tr>
<td>6. Mine the alphaN set</td>
</tr>
<tr>
<td>III. Consolidate and interpret results</td>
</tr>
<tr>
<td>7. Apply the junta algorithm.</td>
</tr>
<tr>
<td>8. Apply the assign-files algorithm</td>
</tr>
<tr>
<td>9. Apply the form-hierarchy algorithm</td>
</tr>
<tr>
<td>10. Apply the subsys-merger algorithm</td>
</tr>
<tr>
<td>11. Present the subsystem decomposition</td>
</tr>
</tbody>
</table>

Figure 3. The steps of the ISA methodology

I. Build a database view of the system. The database view of the system consists of two sets of tuples, called the alphaT set and the alphaN set. The first step to produce these sets consists of analyzing the subject system to identify its programs and files. Then, the sets of programs and files are filtered to eliminate programs and files that cannot form associations. The resulting set of programs and files is known as the alpha set. Finally, the alphaT and alphaN sets are created. The construction of the database view of the system consists of the following steps:
1. Parse the source code. All the programs and files in the system are identified. For each program, a list containing all the files that the program uses is produced.

2. Assign unique identifications. A unique id is assigned to each different program and to each different file in the system to facilitate the manipulation of programs and files.

3. Generate the alpha set. The alpha set is a subset of the programs and files identified in step 1. The alpha set contains the programs and files that have a high probability of forming associations. In other words, this step is a data cleaning process to eliminate noise.

4. Build the alphaT and alphaN sets of tuples. The subsystem identification process requires mining these two data sets. The alphaT set is used to guide the subsystem decomposition, and the alphaN set is used to identify the hierarchies of file implications.

II. Perform data mining. This phase consists of mining associations from the alphaT and the alphaN sets. Only 2-dimensional associations are mined. A 2-dimensional association has the form \( s[p, q] \) where \( s \) is the support of the association, and \( p \) and \( q \) are either programs (if alphaT is used) or files (if alphaN is used). The outcome of this phase consists of two sets of 2-dimensional associations where each set corresponds to one of the input sets. In both cases, the minimum required support is 2. Steps 5 and 6 are:

5. Mine the alphaT set. Use a data mining algorithm to produce 2-dimensional associations. This step produces associations that relate two programs using a common set of files. For example, an association that results from mining the alphaT set may be \( 15[34, 78] \). This association means that programs 34 and 78 use 15 files in common.

6. Mine the alphaN set. Use a data mining algorithm to generate 2-dimensional associations. The associations mined in this step relate two files. Here, the
interpretation is different. The associations are used to create association rules. For example, if a mined association is 10[17, 41] and the confidence of the association rule 41 → 17 is large, then this information is used to create a hierarchy of file implications. In this particular case, the implication means that if a program uses file 41 it also uses file 17.

III. Consolidate and interpret results. In this phase, the two sets of associations produced in phase II are used to guide the clustering process to produce the hierarchical subsystem decomposition as defined in section 3.2.2. This process is a bottom-up approach. First, the associations mined from the alphaT set are used to form groups of programs. Second, the groups are hierarchically organized in several trees (i.e., a forest). Then, the files in the alpha set are assigned to groups to form subsystems. Next, the associations mined from the alphaN set are used to create the hierarchies of file implications. Then, the main subsystems are merged to form a single hierarchy of subsystems (i.e., supra subsystems). Finally, all the information produced in these steps is represented in two representation models. The TO-ISA (Text Output of ISA) model represents the information using a text format. The RM (Representation Model) representation provides a graphical view of the results. The remaining five steps are:

7. Apply the junta algorithm. This algorithm has two stages. First, the mined associations from the alphaT set are used to guide a clustering process that forms groups of programs. Second, the associations guide a merging process in which the groups produced in the first stage are joined to form larger groups. The result of this process is a series of trees. Each tree is a hierarchy of groups. This step, the main part of the subsystem identification process, defines the structure of the subsystem decomposition. Leaf nodes in the trees represent primitive subsystem, non-leaf nodes represent complex subsystems, and root nodes represent main subsystems. However,
the nodes in the trees (i.e., groups of programs) are not yet subsystems because they do not contain files.

8. Apply the *assign-files* algorithm. Each file \( f \) in the alpha set is assigned to the group of programs (i.e., a node in the forest) that contains more programs using \( f \). After applying the assign-files algorithm, the groups contain programs and files. The groups are subsystems according to the definition of subsystems adopted in this research. At this step, the subject system has been decomposed into a hierarchy of data cohesive subsystems.

9. Apply the *form-hierarchy* algorithm. Some of the files in the alpha set cannot be assigned to any particular group of programs because these files are used in many groups. These common files may form hierarchies of file implications. In this step, the associations mined from the alphaN set are used to identify hierarchies of file implications among the common files.

10. Apply the *subsys-merger* algorithm. In this step, a merging algorithm is used to form a single hierarchy of subsystems. The intent is to incorporate in a single hierarchical tree all main subsystems. This step uses a merging algorithm that joins main subsystems to form larger subsystems (i.e., supra-subsystems). This merging process continues until a single supra system is produced.

11. Present the subsystem decomposition. Two representation models are defined in this research. One model is text-based and the other is graphic-based. The text-based model is called the TO-ISA model. TO-ISA is a report-like representation of the subsystem decomposition. The graphical model is called RM. It uses diagrams to represent the subsystem decomposition and the interrelationships among the system components. RM is described in Chapter 4.
Sections 3.3, 3.4, and 3.5 explain in detail each of the three phases of ISA respectively. We first introduce some notation and definitions.

- Greek letters represent parameters, lower case letters are variables, upper case letters are sets or functions, and names in bold are functions.
- The pseudocode used to describe the algorithms is C like. For example, the keyword CONTINUE means that the control of the algorithm goes to the next iteration.
- System $(\mathcal{S})$. \( \mathcal{S} = \{ \mathcal{P}, \mathcal{F} \} \). \( \mathcal{S} \) is a software system composed of a set of programs \( \mathcal{P} \) and a set of data files \( \mathcal{F} \).
- Uses relation. A program \( p \) uses a file \( f \) or \( f \) is used by \( p \) if \( p \) reads or writes information on \( f \).
- \( U(p, F) \) function. \( U(p, F) = | \{ f \in F : p \text{ uses } f \} | \). The \( U \) function gives the number of files in a set \( F \) that a program \( p \) uses.
- \( Q(f, P) \) function. \( Q(f, P) = | \{ p \in P : p \text{ uses } f \} | \). The \( Q \) function gives the number of programs in a set of programs \( P \) that use file \( f \).
- Tuple \((t)\). A tuple \( t \) is a set of items \( \{ x_1, x_2, \ldots \} \) where \( t \subseteq \mathcal{P} \) or \( t \subseteq \mathcal{F} \).
- Frequency \((fq(x))\). The frequency of an item \( x_i \) denoted by \( fq(x_i) \) is the number of tuples in a given set of tuples \( T \) that contain \( x_i \).
- Association \((a)\). An association \( a \) has the form \( a = s[x_i, x_j] \). The association means that there are \( s \) tuples in a given set of tuples \( T \) that contain the items \( x_i \) and \( x_j \). \( s \) is called the support.
- Confidence \((c)\). The confidence is defined for each element of an association. It is obtained by dividing the support of the association by the frequency of the element. For example, for the association \( s[x_i, x_j] \), \( c_i = s/fq(x_i) \) and \( c_j = s/fq(x_j) \). Thus, \( c_i \) and \( c_j \) are the confidences of \( x_i \) and \( x_j \) respectively.
3.3. Build a Database View

The objective of this phase is to produce a database representation of the subject system. The representation produced in this step was derived from the input requirements of the data mining algorithm used in the next phase. The input to this type of mining is well defined. It is a set of transactions or tuples. Therefore, the database view is a set of tuples.

In particular, this phase produces two sets of tuples, named the alphaT set and the alphaN set. The former set is used to break down the subject system into subsystems while the later set is used to identify hierarchies of file implications. The generation of the alphaT and alphaN sets is done in four steps: parse the source code, assign unique identifications, generate the alpha set, and build the alphaT and alphaN sets. Sections 3.3.1 to 3.3.4 describe each of these steps.

3.3.1 Parse the source code

The source code is parsed to produce the list of programs and the list of files in the target system. In addition, this step produces the relation “p uses f” where p is a program, f is a file, and uses is the relation defined in the end of the previous section. Thus, the input to this step is the source code of the subject system and the output is the triplet \( \langle P, \mathcal{F}, R \rangle \) where \( R \subseteq (P \times \mathcal{F}) \) and \( R = \{ (p, f) | p \in P, f \in \mathcal{F}, \text{ and } p \text{ uses } f \} \).

The identification of programs and files can be done with a parser. This parser has to be customized for the particular language used to build the subject system, and for the definition of “program” and “file” adopted in the analysis. For example, the parser may need to be capable of processing calls to other source-code files such as “copy” files in COBOL or embedded code such as SQL code.

3.3.2 Assign unique identifications

This step consists on assigning a unique identifier to each program and to each file. For example, if \( n \) programs are identified (i.e., \( |P| = n \)), then the programs ids are 1, 2, 3, ..., \( n \).
Similarly, if there are $m$ files, they are labeled 1, 2, 3, ..., $m$. For clarity reasons, the notation $px$ is used to denote the id of program $x$, and $fy$ denotes the id of file $y$. For instance, $p3$ and $f4$ are the ids of program 3 and file 4, respectively. In addition, the relation $R$ has to be re-coded using the identifiers. Thus, the input to this step is the triplet generated in the previous step, and the outputs are three lists. The first list is a list of tuples <p-id, p-name>, where p-id is the program id and p-name is the program name. Similarly, the second list is a list of tuples <f-id, f-name> where f-id is the file id and f-name is the file name. The last list is a list of tuples <p-id, f-id> which is the coded relation. This labeling of programs and files is required to facilitate the coding and processing of the information.

3.3.3 Generate the alpha set

This step is a data preprocessing process. Its objective is to generate a "clean" data set for the mining process. The resulting data set is known as the alpha set. The alpha set is the set $A=\{P, F\}$ such that $P \subseteq \mathcal{P}$, $F \subseteq \mathcal{F}$, $P = \{p \mid U(p, F) > \gamma\}$, and $F = \{f \mid Q(f, P) > \beta\}$ where $\gamma, \beta$ are integers and $\gamma > 0$, $\beta > 0$. The alpha set contains programs that use more than $\gamma$ files and files that are used by more than $\beta$ programs. These parameters are necessary to avoid introducing noise to the analysis. For example, if $\gamma = 0$ were allowed, the alpha set would include programs that use just one file. These programs do not provide information to form associations among files because an association requires at least two components. A similar argument applies in the case of files. Thus, the alpha set is built to assure that the data mining algorithm finds associations.

To produce the alpha set, it is not enough to remove from $\mathcal{S}$ the programs and files that do not satisfy the constraints $U(p, F) > \gamma$ and $Q(f, P) > \beta$. It is possible that by removing a program or file from $\mathcal{S}$ another program or file will not satisfy these constraints. Thus, the generation of the alpha set is an iterative process. In each iteration the programs and files that
do not satisfy the $\gamma$ and $\beta$ constraints are removed from the final set of programs and files.

This process is repeated until no program or file is removed. These ideas are incorporated in the algorithm that produces the alpha set, shown in Figure 4.

**INPUT:**
(a) The set of programs $\mathcal{P}$
(b) The set of files $\mathcal{F}$

**OUTPUT:**
The alpha set $A = \{P, F\}$, $P \subseteq \mathcal{P}$, $F \subseteq \mathcal{F}$, $P = \{p \mid \text{U}(p, F) > \gamma\}$, $F = \{f \mid \text{Q}(f, P) > \beta\}$, and $\gamma$, $\beta$ are integers and $\gamma > 0$, $\beta > 0$.

1. done = false, $F = \emptyset$, $P = \mathcal{P}$, $P' = \emptyset$, $P' = \emptyset$
2. DO
3. $F' = \{f \in F \mid \text{Q}(f, P) > \beta\}$
4. $P' = \{p \in P \mid \text{U}(p, F') > \gamma\}$
5. IF ($F' = F$) AND ($P' = P$) THEN
6. done = true
7. ELSEIF
8. $F = F'$
9. $P = P'$
10. UNTIL (done)
11. $A = \{P, F\}$

Figure 4. Algorithm to produce the alpha set

It is possible to use constants instead of the parameters $\gamma$ and $\beta$ (i.e., $\gamma = 1$, and $\beta = 1$). Indeed, the use of $\gamma = 1$ and $\beta = 1$ is enough to assure that the data mining algorithm finds associations. However, it is more flexible to use the parameters $\gamma$ and $\beta$. For instance, for large systems it may be better to use larger values for these parameters if the system contains many files that are used by just two programs. In this case, these files will produce many associations that do not provide any significant information. On the contrary, these associations may introduce noise to the subsystem identification process.

3.3.4 Build the alphaT and alphaN sets

The objective of this step is to produce the alphaT and the alphaN sets, which represent the database view of the subject system. Both sets are based on the alpha set. Since the alphaT set is used to guide the subsystem decomposition, it is designed to produce associations that
relate programs. The alphaN set is used to identify the hierarchies of file implications; thus, it is designed to produce associations rules (i.e., implications of the form \( f \rightarrow q \), where \( f, q \in F \)).

Let \( A = \{P, F\} \) be the alpha set where \( P = \{p_1, p_2, ..., p_{|P|}\} \), \( F = \{f_1, f_2, ..., f_{|F|}\} \). Then, the alphaT set is a set of file-based tuples. That is, there is a tuple for each file in the alpha set. The tuple corresponding to file \( f \) contains the ids of all the programs that use \( f \). Formally, the alphaT set is a relation defined as follows: \( \alpha_T \subset (P \times F) \) such that \( \alpha_T = t_1, t_2, ..., t_{|P|} \) where \( t_i = \{ p \in P \mid p \text{ uses } f_i \} \).

Similarly, the alphaN set is a set of program-based tuples. In this case, there is a tuple for each program in the alpha set. The tuple corresponding to program \( p \) contains the file ids of all the files that \( p \) uses. Formally, \( \alpha_N \subset (P \times F) \) such that \( \alpha_N = t_1, t_2, ..., t_{|P|} \) where \( t_i = \{ f \in F \mid p_i \text{ uses } f \} \).

The alphaT and alphaN sets can be viewed as matrices. AlphaT can be seen as a matrix in which each row represents a file and each column a program. AlphaN can be seen as the transpose of the matrix representing AlphaT. In both cases, the matrix is formed by giving a value of one when the program represented by a row/column uses the file represented by the column/row and a value of zero otherwise.

3.4. Perform Data Mining

The objective of this phase is to mine associations from the alphaT and the alphaN sets. Particularly, the specific objective is to generate 2-dimensional associations from these sets of tuples. A 2-dimensional association has the form \( s[p, q] \) where \( s \) is the support of the association, and \( p \) and \( q \) are either programs (if alphaT is used) or files (if alphaN is used).

This objective has some implications. First, the objective of the mining process is not to mine association rules but associations. An association rule is an implication \( p \rightarrow q \) (\( c\% \)), where \( p \) and \( q \) are elements of the input set and \( c \) is the confidence of the rule. The relationship between the association rule and the association is that the latter is used to produce the former.
For example, the association $s[p, q]$ produces the following two association rules $p \rightarrow q$ (c$_1$%) and $q \rightarrow p$ (c$_2$%). This distinction comes from a family of algorithms that mine association rules described in [Agra93]. These algorithms are designed to work in two stages. In the first stage, they find $k$-dimensional associations ($k = 1, 2, \ldots$) for all possible values of $k$. In each dimension they find all associations that satisfy predetermined minimum values for confidence and support. These associations are called large itemsets. Thus, in the first stage, these algorithms find large $k$-itemsets. The largest possible value for $k$ is determined by the data set. Thus, $k$ is the largest possible set of elements that can form an association in that particular data set and that satisfy the requirements for confidence and support.

Once the large $k$-itemsets are identified, the second stage consists of generating the association rules. This generation is straightforward; it works as the example in the previous paragraph. The only difference is that for $k$-itemsets, where $k > 2$, all the combinations of the elements of the itemset (association) have to be checked against the required confidence value. For example, if $s[p, q, r]$ is one of the large 3-itemsets found, then many association rules can be produced such as $p \rightarrow \{q, r\}$ (c$_1$%), $q \rightarrow \{p, r\}$ (c$_2$%), $\{r, p\} \rightarrow q$ (c$_3$%). From these association rules, only the rules with confidence larger than the required confidence threshold are considered. A method to calculate the confidence of an association rule is given in section 2.1.

Consequently, this data mining phase has to use an algorithm from the family of algorithms described in [Agra93]. The algorithm has to be customized to produce only large 2-itemsets (i.e., 2-dimensional associations). Finally, algorithms to mine association rules require two input parameters: the minimum required support and confidence. This phase does not require a minimum value for confidence because finding large item sets does not require a confidence threshold, but it requires a minimum value for support, denoted by $\sigma$. Thus, the
mining process has to produce just 2-dimensional associations that have support that is equal or greater than $\sigma$.

The data mining algorithm is run twice. Indeed, there are two separate data mining processes, one for the alphaT and one for the alphaN. These processes are done in steps 5 and 6 respectively.

In step 5, the alphaT set is mined to identify 2-dimensional associations. Let $A =\{P, F\}$ be the alpha set such that $P =\{p_1, p_2, \ldots, p_{|P|}\}$, and $F =\{f_1, f_2, \ldots, f_{|F|}\}$. The input to step 5 is the alphaT set given by $\alpha_T = t_1, t_2, \ldots, t_{|P|}$ where $t_i = \{p \in P | p \text{ uses } f_i\}$. The outcome of the mining process is the T-List given by $T-List = (a_1, a_2, \ldots, a_n)$ where $a_i = s[p, q]$ is an association, $p \in P$, $q \in P$, and $s$ is the support of the association such that $s \geq \sigma$.

Step 6 follows a similar procedure. In this step, the alphaN set is mined to identify 2-dimensional associations. The input to this step is the alphaN set given by $\alpha_N = t_1, t_2, \ldots, t_{|P|}$ where $t_i = \{f \in F | p_i \text{ uses } f\}$. The output of this step is the N-List given by $N-List = (a_1, a_2, \ldots, a_m)$ where $a_i = s[f, g]$ is an association, $f \in F$, $g \in F$, and $s$ is the support of the association such that $s \geq \sigma$.

In summary, the data mining phase of the ISA methodology consists on applying a data mining algorithm to the alphaT and to the alphaN sets to produce large 2-itemsets that have support greater than or equal to $\sigma$. The outcome of this phase consists on two lists of associations, the T-List obtained from mining the alphaT set, and the N-List generated from the alphaN set.

### 3.5. Consolidate and Interpret Results

The objective of this phase is to produce the hierarchical subsystem decomposition of the subject system based on the mined associations. This phase include five steps (steps 7 to 11 of...
the ISA methodology). The inputs to this phase are the T-List and N-List generated in phase II. The high-level algorithm for this phase is as follows:

1. Form groups of programs
2. Organize these groups hierarchically
3. Assign files to these groups (after that, the groups become subsystems)
4. Form hierarchies of file implications using common files
5. Merge main subsystems to create a single tree of subsystems
6. Represent the subsystem decomposition

The *junta* algorithm in step 7 performs the functions in lines 1 and 2. The *assign-files*, *form-hierarchy*, and *subsys-merger* algorithms compute the functions in lines 3 to 5 respectively. Last, there are two options to represent the outcome of the ISA methodology. The first option is a text-based output called TO-ISA. The other option is a graph-based representation called RM.

3.5.1 The junta algorithm

The *junta* algorithm (junta means join in Spanish) implements the main objective of this phase, namely the decomposition of the subject systems into subsystems. The input to the junta algorithm is the T-List produced in phase II. The output of junta is a set of groups of programs, which are hierarchically organized.

Junta can be divided in two stages. In the first stage, junta uses the associations in the T-List to guide a clustering process whose objective is to organize the programs in the alpha set into several disjoint groups (sets) of programs. In the second stage, junta uses some of the associations in the T-List to guide a process that organizes the groups of programs into several hierarchies. Hence, the output of the junta algorithm is a series of hierarchical trees (i.e., a forest) whose nodes are groups of programs.
These sets of programs are referred to as groups instead of as subsystems because they do not contain files. According to the definition of subsystem adopted in this research, a subsystem is a set of programs and a possible empty set of files. Thus, since the files have not been assigned to the groups identified by junta, these sets of programs are not considered subsystems yet. Nevertheless, the hierarchical trees that junta produces define the structure of the final subsystem decomposition.

The junta algorithm has been divided to facilitate its explanation. The complete algorithm is shown in Figure 5 to Figure 10. Figure 5 is the header of the algorithm. It introduces some notation, and describes the input and the output of junta. Figure 6 shows the pseudocode for the first stage of junta. Figure 8 presents the pseudocode for the second stage. Finally, Figure 7, Figure 9, and Figure 10 show the pseudocode of three subroutines of junta: is-member, merge, and is-subset. A detailed explanation of each part of junta follows.

The header of the algorithm contains three parts: definitions, input, and output (Figure 5). The first part introduces some notation and definitions. Most of these definitions are self-explanatory, however, the last two need more elaboration. First, the function parent(G) is required to track the structure of the groups when they are organized in hierarchies. In a hierarchy, each non-root node (group) has a father group. Thus, parent(G) denotes the group of which G is a child. Second, the core of a group, denoted by \( G^* \), is a subset of programs in G (i.e., \( G^* \subseteq G \)). \( G^* \) is the subset of programs that originated the group. In other words, the programs in the core of a group are the signature of the group.

The input of the algorithm consists of 5 elements: the T-List that is used to guide the grouping, an empty list that is used as an intermediate data store, and three parameters that control the behavior of the junta algorithm. The T-List has to be sorted according to 5 criteria. These sorting criteria are inclusive. For example, if there are many associations with the same value for the product \( e_p e_q \), then these associations are sorted in descending order by support.
which is the second sorting criterion. Consequently, the notation \((a_1, a_2, \ldots, a_n)\) implies that the product of \(c_p \cdot c_q\) in the association \(a_i\) is greater or equal than the product \(c_p \cdot c_q\) in association \(a_j\) for all \(i \leq j\). Finally, the output part of the junta header formally describes the outcome of junta.

**Junta algorithm. Version 2.1.**

**DEFINITIONS:**
- Let \(G, H\) be sets of programs, or groups, such that \(G \cap H = \emptyset\).
- Let \(g(p)\) denote the group that contains program \(p\).
- Let \(\text{parent}(G)\) denote the parent set (group) of \(G\).
- Let \(G^*\) be the programs that form the core of the group \(G\).

**INPUT:**
(a) T-List = \((a_1, a_2, \ldots, a_n)\) sorted by \((c_p \cdot c_q)\) (descending), then by \(s\) (descending), then by \(\max(c_p, c_q)\) (descending), then by \(p\) (ascending), finally by \(q\) (ascending). \(a_i = s[p, q]\) and \(c_p, c_q\) are the confidences of \(p, q\) respectively, where \(p, q \in P\) and \(P\) is the set of programs in the alpha set.
(b) \(L_1 = \{\}\)
(c) \(0 < \delta < 1. \ [\delta, 1]\) defines the range of accepted values for \((c_p \cdot c_q)\). Default value of this parameter: \(\delta = 0.5\).
(d) \(0 < \tau < 0.5\). \(\tau\) is the percentage of files within a program allowed to be new in a group when the program enters the group. Default value \(\tau = 0.3\).
(e) \(0 < \epsilon < 0.5\). \(\epsilon\) is the percentage of files allowed to be used once within a group. Default value \(\epsilon = 0.4\).

**OUTPUT:**
A decomposition of \(P\) in \(h\) sets (groups) \(Z = \{Z_1, Z_2, \ldots, Z_h\}\) where \(Z_i\) is a tuple \(<G_i, ind_i>\), \(G_i \subseteq P\); \(ind_i = \text{null or } ind_i \in \{1, 2, \ldots, h\}\) and \(ind_i \neq i\); besides, \(G_i \cap G_j = \emptyset\) for \(i \neq j\). However, \(G_1 \cup G_2 \ldots \cup G_h\) may not be equal to \(P\). \(ind_i\) is the index of the set that is parent of set \(Z_i\). For example, if \(Z_4\) is parent of \(Z_6\), then \(ind_6 = 4\). If \(ind_i = \text{null}\) then \(Z_i\) is a family root. Thus, the output of the junta algorithm is a hierarchical decomposition of \(P\) in \(h\) disjoint subsets organized in several trees rooted in certain elements of \(Z\) known as family roots.

Figure 5. Header of the junta algorithm

Figure 6 contains the pseudocode of the first stage of junta. The objective of this stage is to organize the programs in the alpha set into several disjoint sets, called groups. The grouping criterion is that the programs are grouped according to the files they use. The programs that use a similar set of files should be together in a group. In other words, the groups should be data cohesive in the sense that the programs in a group use a similar set of data repositories.
The key idea here is that associations provide the required information to form these groups. For example, consider the association $a_i = s[p, q]$. The interpretation of this association is that programs $p$ and $q$ use the same $s$ files. It does not mean that $p$ or $q$ use just $s$ files, rather it means that $s$ files are common between $p$ and $q$. If $s$ is large compared to the number of files used by $p$ or $q$, then these programs can form a group. However, if $s$ is just a fraction of the files used by $p$ or $q$, then these programs do not share many files; thereby, they do not form a group.

An incremental approach is used to form the groups of programs. The underlying idea for this process is to take one association at a time, and form a group with the two programs contained in the association. If one of the programs is already in one group, then the algorithm uses a similarity measure to decide whether the other program can be incorporated to the group or not. This process is repeated until all the associations are analyzed. Thus, a group is created with the two programs in an association. These two originating programs are called the core of the group. Then, the group accepts more programs according to some similarity functions.

The sorting order defined in Figure 5 for the input list of associations (i.e., T-List) is crucial for the success of this approach. There are two reasons for using those five sorting criteria. First, these criteria assure that the associations are always in the same order regardless of the sorting algorithm used to sort them. This feature assures repeatability of the analysis. Indeed, ISA produces the same results if it is fed with the same parameters. Second, this sorting order is fundamental for the subsystem decomposition process. The key idea is that the associations with largest confidence in both directions come first in the sorted T-List. These associations contain the programs that use the same set of files. Thus, the product of the confidences of the two programs in the association is selected as the main sorting criteria.
Junta 2.1

1. FOR (i = 1; i ≤ n AND $\max(c_p, c_q) \geq \delta$; i++)
   
   Let $a_i = [p, q]$

2. IF ($g(p) = \text{null AND } g(q) = \text{null}$) THEN
   
   $Z_{new} = \langle G, \text{null} \rangle$, $G = G^* = [p, q]$
   
   CONTINUE

3. IF ($g(p) = g(q)$) THEN
   
   CONTINUE

4. IF ($g(p) \neq \text{null AND } g(q) = \text{null}$) THEN
   
   Assume $G = g(p)$

   IF ($c_q = 1$) THEN
   
   $G = G \cup \{q\}$

   IF (($c_p = 1$ AND $p \in G^*$) THEN
   
   $G^* = G^* \cup \{q\}$

   ELSE
   
   IF (is-member($q, G, \tau, \varepsilon$)) THEN
   
   $G = G \cup \{q\}$

   CONTINUE

   ENDIF

5. IF ($g(p) = \text{null AND } g(q) \neq \text{null}$) THEN
   
   Assume $G = g(q)$

   IF ($c_p = 1$) THEN
   
   $G = G \cup \{p\}$

   IF (($c_q = 1$ AND $q \in G^*$) THEN
   
   $G^* = G^* \cup \{p\}$

   ELSE
   
   IF (is-member($p, G, \tau, \varepsilon$)) THEN
   
   $G = G \cup \{p\}$

   CONTINUE

   ENDIF

6. IF ($g(p) \neq \text{null AND } g(q) \neq \text{null AND } g(p) \neq g(q)$) THEN
   
   IF ($c_p = 1$ OR $c_q = 1$) THEN
   
   $L_i = L_i + a_i$

ENDFOR

FOR ($j = i + 1; j \leq n; j++$

Let $a_i = [p, q]$

IF ($\max(c_p, c_q) = 1$) THEN

$L_i = L_i + a_j$

ENDIF

ENDFOR

Figure 6. Pseudocode of the first stage of the junta algorithm

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
For example, consider the association $a_i = s \_ [p, q]$. $c_p = slf(p)$ and $c_q = slf(q)$. The largest value for $c_p * c_q$ is 1. The only way to obtain this value is if both $c_p$ and $c_q$ are equal to 1. This means that $p \rightarrow q$ with 100% of confidence and that $q \rightarrow p$ with 100% of confidence. In this sense, this association has a 100% confidence in both directions. The meaning of associations with large confidence in both directions is that the programs in the association use the same set of files. This is a key step to forming clusters of programs. Hence, the first sorting criterion assures that the associations containing strongly related programs appear first. If there are many associations with the same value for $c_p * c_q$, then the second sorting criterion assures that the association with larger support appears first. In other words, programs using the largest amount of common files appear first. Indeed, groups with core programs that have a large set of common files (i.e., support) are preferred over groups with core programs having a small set of common files. A large set of common files promotes the incorporation of more programs to the group, which leads to larger groups of programs.

The third sorting criterion is the maximum of the confidences. This criterion is used to assure that the associations with one large confidence are analyzed first. For example, a value of $c_p * c_q = 0.56$ can be produced by $(0.7 * 0.8)$ or by $(1 * 0.56)$. The former implies that $p$ and $q$ use some files in common, and the later implies that $p$ uses a subset of the files that $q$ uses. The second option is preferred to form groups because all the files that $p$ uses are also used by $q$. Hence, the criterion $\max(c_p, c_q)$ puts the second option first since $\max(1, 0.56) > \max(0.7, 0.8)$. The last two sorting criteria are included to assure that the order of the associations in T-List is the same regardless of the sorting algorithm used to sort T-List.

Before describing in detail the junta algorithm, some definitions are required. Let $\mathcal{F}(p)$ denote the set of files that program $p$ uses, $\mathcal{F}(G)$ the set of files that the programs in group $G$ use, and $\mathcal{U}(G)$ the set of files that are used by only one program in group $G$. For example, assume that program $p$ uses files $a, b, c,$ and $d$, that program $q$ uses files $b, c, d$ and $e$, and that
\( G = \{ p, q \} \). Thus, \( \mathcal{H}(p) = \{ a, b, c, d \}, \mathcal{H}(q) = \{ b, c, d, e \} \), \( \mathcal{H}(G) = \{ a, b, c, d, e \} \), and \( \mathcal{U}(G) = \{ a, e \} \). In the case that \( G \) belongs to a hierarchy of groups, \( \mathcal{H}(G) \) include all the files used by all the programs assigned to all the nodes in the tree rooted in \( G \). In addition, the following definition is introduced.

Definition. The component \( A \) covers the component \( B \) if \( \mathcal{H}(B) \subseteq \mathcal{H}(A) \). \( A \) and \( B \) can be either a program or a group.

The first stage of the junta algorithm works as follows. A loop is used to traverse the T-List. In each iteration, a single association is analyzed (lines 1 to 33 in Figure 6). The parameter \( \delta \) defines the range \( \delta \leq (c_p * c_q) \leq 1 \). It delimits the associations in the T-list that are analyzed. This parameter is necessary to avoid analyzing associations with low confidences. Low confidences mean that the programs in the associations do not use many files in common. Empirically, we have observed that if \( \delta < 0.5 \) the associations do not provide significant information. When \( c_p * c_q \) is less than 0.5, it means that at least one of the confidences is less than 50%. This level of confidence implies that less than 50% of the files that a program uses are used by the other program in the association. These programs do not form a group. On the other hand, larger values of \( \delta \) produce groups of programs that use almost the same set of files.

The analysis of each association has four possibilities (lines 3 to 32):

1. Both programs in the association have not been assigned to any group (lines 3 to 5).
   In this case, the programs in the association form a new group. In addition, these two programs form the core of the group
2. Both programs are already in the same group (lines 6 and 7). No action is necessary.
3. One program is assigned to one group and the other is not (lines 8 to 29). Lines 8 to 18 and lines 19 to 29 do the same function. The first set of lines is for the case that program \( p \) is assigned to a group and \( q \) is not. The second set of lines is for the opposite case. Assume that \( q \) is not assigned to any group and \( p \) is assigned to group
If the confidence of \( q \) is 100%, then all the files that \( q \) uses are used by \( p \). In other words, \( p \) covers \( q \), thereby \( G \) covers \( q \). Thus, program \( q \) enters the group \( G \) (lines 10 and 11). If in addition the confidence of \( p \) is 100% and \( p \) is in the core of \( G \), then this means that \( q \) and \( p \) use the same set of files. Thus \( q \) becomes part of the core of \( G \) (lines 12 and 13). However, if the confidence of \( q \) is less than 100%, then it is necessary to evaluate whether \( q \) can be part of \( G \). The is-member function is used to make the evaluation (lines 14 to 18).

(4) The programs in the association are assigned to different groups (lines 30 to 32). This result indicates that the programs in the association are already covered by programs in different groups. Nevertheless, it also may indicate that the programs belong to groups that are hierarchically related. An example of this case is when \( g(p) \) covers \( g(q) \). If this is the case (i.e., one of the confidences is 100%), then the association is saved in the temporary list \( L_I \). This list is used in the second stage of the junta algorithm to define the hierarchical organization of the groups.

Finally, lines 34 to 39 complete the traversal of the sorted \( T \)-List and add to \( L_I \) the associations that contain at least one confidence equal to 100%. This step assures that associations in which one of the programs covers the other program (i.e., 100% confidence) are considered for the next stage. In addition, this mechanism assures that programs using few files have an opportunity to be included in some group.

The is-member function (Figure 7) determines if by adding a new program \( q \) to a group \( G \), the internal cohesiveness of \( G \) is maintained. If it is, \( q \) is included in \( G \). The two criteria for this decision are based on coverage (line 2). First, \( G \) has to cover most of the files that \( q \) uses. This constraint assures the data cohesiveness within the group. The ratio between the non-covered files and the total files that \( q \) uses serves as a measure for this criterion. The \( \tau \) parameter defines the range of accepted values for this ratio (i.e., \( 0 < \tau < 0.5 \)). Clearly, for
small values of τ the programs that join a group need to have a high percentage of covered files. The second criterion minimizes the number of files in $\mathcal{F}(G)$ that are used by just one program. A ratio is also used to measure this criterion. The $\varepsilon$ parameter defines the range of accepted values for the ratio (i.e., $0 < \varepsilon < 0.5$). This criterion is necessary because the presence of few files in a group that are used by just a few programs in the group is as important as the presence of many files used by most of the programs in the group.

is-member(q: program, G: group, τ: real, ε: real)
Let $U(G) =$ files that are used once in group $G$
$\mathcal{F}(G) =$ Set of files used by programs in group $G$ (Include files used by programs in the children groups of $G$)
$\mathcal{F}(p) =$ Set of files used by program $p$
1 $G' = G \cup \{q\}$
2 IF $\left( \left( |\mathcal{F}(q)| - |\mathcal{F}(G)| \right) / |\mathcal{F}(q)| \right) \leq \tau \) AND $\left( \left( |U(G')| / |\mathcal{F}(G')| \right) \leq \varepsilon \right) \) THEN
3 is-member = true
4 ELSE
5 is-member = false
6 END is-member

Figure 7. Pseudocode of the is-member function

A final note on the first stage of the junta algorithm. This portion of the junta algorithm groups the programs in the alpha set into several groups. It uses groups of size two as the starting point. Then, it adds programs to make the groups larger. However, there is no provision to merge groups in this stage. The reason for this decision is that it is easier to control the internal cohesiveness of a group if no group merging is done. The internal cohesiveness of a group is maintained by using the concept of core programs. Core programs can be seen as the centroid of a centroid-based clustering algorithm. If the method had allowed the merging of groups, then there would have been the problem of deciding which programs would be the core of the merged group. It is more consistent to organize the groups hierarchically rather than to merge them.
SORT(L_1, (c_p^c_q) (D), s (D), p (A), q (A)) (*D=descending, A=ascending*)

FOR(i = 1; i ≤ |L_1| AND MAX(c_p, c_q) = 1; i++)

Let a_i=[p, q]

Assume G = g(p) if g(p) ≠ null
Assume H = g(q) if g(q) ≠ null

IF (p ∈ G^+ AND q ∈ H^+) THEN

IF (c_p = 1) THEN

IF (parent(G) ⊂ H) THEN

merge (H, G, τ)

ELSE

IF (parent(H) ⊂ G) THEN

merge (G, H, τ)

ELSE

IF (g(p) = G AND g(q) = null AND c_q = 1) THEN

G = G ∪ {q}

ELSE IF (g(q) = H AND g(p) = null AND c_p = 1) THEN

H = H ∪ {p}

ENDIFOR

END Junta

Figure 8. Pseudocode of the second stage of the junta algorithm

Figure 8 shows the pseudocode for the second stage of the junta algorithm. The objective of this stage is to organize hierarchically the groups identified in the previous stage. Stage two has two main parts: the sort of L_1 (line 39) and a loop that is used to traverse L_1 (lines 40 to 56). The general idea of this process is to use the associations that contain at least one confidence equal to 100% to guide the formation of hierarchies. The rule is that if a program p in G^+ covers a program in H^+, then H becomes a child of G. The sorting criteria for the L_1 have the same motivation as the criteria used to sort T-List.

This stage also uses a loop to analyze one association in each iteration (line 40). Line 44 is used to assure that both programs are in the core of their respective groups. If this test fails, it is possible that one of the programs in the association has not been assigned to any group. The code between lines 51 and 56 tests this condition. If this condition is detected, then the algorithm assigns the unassigned program to the corresponding group. However, if both programs in the association belong to the core of their groups, then it is necessary to determine
which program covers the other. If \( q \) covers \( p \) then the algorithm continues in lines 46 and 47, otherwise lines 49 and 50 are executed. The former situation means that \( g(p) \) is a child of \( g(q) \). The latter situation means that \( g(q) \) is a child of \( g(p) \).

\[
\text{merge}(H: \text{group}, G: \text{group}, t: \text{real})
\]

Let \( \text{FamilyRoot}(G) \) be the root of the tree that contains \( G \)
Assume \( Z_x = <G, x> \), where \( x \) is either null or an integer
Assume \( Z_y = <H, y> \), where \( y \) is either null or an integer

1. IF (\( \text{parent}(G) = {} \)) THEN
2. \hspace{1em} IF (\( \text{is-subset}(H, G, t) \)) THEN
3. \hspace{2em} \( Z_x = <G, b> \)
4. \hspace{1em} ELSE IF (\( \mathcal{F}(G) \subset \mathcal{F}(\text{parent}(G)) \)) THEN
5. \hspace{2em} Reject. Do not merge, exit procedure
6. \hspace{1em} ELSE IF (\( \mathcal{F}(G) \subset \mathcal{F}(H) \)) THEN
7. \hspace{2em} \( Z_x = <G, b> \)
8. \hspace{1em} ELSE IF (\( \mathcal{F}(G) \subset \mathcal{F}(\text{FamilyRoot}(G)) \)) THEN
9. \hspace{2em} Reject merge, exit procedure
10. \hspace{1em} ELSE IF (\( \mathcal{F}(G) \subset \mathcal{F}(\text{FamilyRoot}(H)) \)) THEN
11. \hspace{2em} \( Z_x = <G, b> \)
12. \hspace{1em} ELSE IF (\( |\mathcal{F}(G) - \mathcal{F}(\text{FamilyRoot}(G))| \geq |\mathcal{F}(G) - \mathcal{F}(\text{FamilyRoot}(H))| \)) THEN
13. \hspace{2em} \( Z_x = <G, b> \)
14. \hspace{1em} ELSE
15. \hspace{2em} Reject. Do not merge, exit procedure.
16. \hspace{1em} ENDIF
17. END \text{merge}

Figure 9. Pseudocode of the merge procedure

Lines 46 and 49 are included for efficiency in that they check if \( g(p) \) and \( g(q) \) are already in a parent-child relationship. If so, the algorithm goes to the next iteration. If the algorithm reaches either line 47 or line 50, then \( g(p) \) and \( g(q) \) relate hierarchically to each other. However, the problem of assigning these groups to the corresponding hierarchy still exists. For example, assume that \( g(p) \) covers \( g(q) \). Then \( g(p) \) is the root of a hierarchy that contains \( g(q) \). In turn, \( g(p) \) may be an internal node of a larger hierarchy and \( g(q) \) may be the root of another hierarchy. The problem becomes finding the right location in the hierarchy rooted in \( g(p) \) to place \( g(q) \). The \text{merge} procedure that is called in lines 47 and 50 solves this problem.
The merge procedure is shown in Figure 9. Assuming that a program in $H^*$ covers a program in $G^*$, the merge procedure works as follows. If $G$ is a hierarchy root or a group that does not belong to any hierarchy (i.e., $G$ has no parent), then $G$ is a candidate for a child of $H$ (line 1). The \textit{is-subset} function (Figure 10) determines if $G$ can be a child of $H$. This test is based on coverage. If $H$ covers $G$, then $G$ becomes a child of $H$. If the opposite is true, then a ratio of files not covered is used to determine if $G$ can still be a child of $H$ (line 4 in Figure 10). This criterion is similar to the criterion used in the is-member function. The test promotes that any group in the hierarchy covers all the groups that are its descendents in the hierarchy tree. This constraint helps to maintain a tight data cohesiveness within the hierarchies.

\begin{verbatim}
 is-subset (H:group, G:group, \tau:real)
 1 IF (\mathcal{F}(G) \subseteq \mathcal{F}(H)) THEN
 2    is-subset = true
 3 ELSE
 4     IF (|\mathcal{F}(G) - \mathcal{F}(H)| / |\mathcal{F}(G)| \leq \tau ) THEN
 5        is-subset = true
 6     ELSE
 7        is-subset = false
 8 END is-subset
\end{verbatim}

Figure 10. Pseudocode of the is-subset function

If $G$ has already a parent, then this relationship takes precedence over $H$ (lines 4 and 5). However, if the parent of $G$ does not cover it but $H$ does, then $G$ becomes a child of $H$ (lines 6 and 7). It is important to notice that $G$ may be the root of a tree. When $G$ becomes a child of $H$ the complete tree of which $G$ is the root becomes part of the tree rooted in $H$. If all these conditions fail, then the coverage is tested against the family roots of groups $G$ and $H$ (lines 8 to 17) to find the best place in the hierarchy to locate $G$. The family root of a group $G$ is the group $X$, such that $X$ belongs to the same hierarchy as $G$ and $X$ has no parent (e.g., $X$ is the root group of the hierarchy that contains $G$).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
3.5.2 The assign-files algorithm

Once the programs in the alpha set have been organized hierarchically in groups, the next step is to assign the files in the alpha set to these groups. Once the groups contain both programs and files, they are called subsystems. The assign-files algorithm carries out this file assignment.

Let \( f_q(f) \) be the number of programs in the alpha set that use file \( f \). The criterion to assign a file \( f \) to a group \( G \) is that \( G \) must have at least \( \eta f_q(f) \) programs that use \( f \), where \( 0.5 \leq \eta < 1 \). For example, if \( \eta = 0.8 \) then a file \( f \) is assigned to group \( G \) if \( G \) contains at least 80% of the programs in the alpha set that use \( f \). It is important to notice that \( G \) can be a non-leaf group (i.e., an internal node in a hierarchy of groups). In this case, the file assignment process considers all the programs in all the groups belonging to the tree rooted in group \( G \). This provision allows that files that are used proportionally by several groups are assigned to the parent of those groups.

![Z data structure](image1)

\[ Z \]  

\begin{align*}
Z &\quad \text{data structure} \\
1 &\quad \text{Primitive subsystems} \\
2 &\quad \text{Complex subsystems}
\end{align*}

![S data structure](image2)

\[ S \]  

\begin{align*}
S &\quad \text{data structure} \\
A &\quad \text{Primitive subsystems} \\
1 &\quad \text{Complex subsystems}
\end{align*}

**Figure 11. Example of transforming \( Z \) into \( S \)**
To use this criterion it is necessary to adapt the $Z$ data structure generated by the junta algorithm (see the output section in Figure 5). The new data structure is called the $S$ data structure. The transformation of $Z$ into $S$ consists on making each parent node a sibling of its children. An example of such a transformation is given in Figure 11. The left part of the figure contains an example of a $Z$ data structure and the right part is the corresponding $S$ data structure. The squares labeled with numbers represent groups of programs. The figure shows that $S$ contains additional nodes (i.e., the nodes labeled with letters). These nodes will become complex subsystems and the original nodes will become primitive subsystems. The $Z$-to-$S$ algorithm shown in Figure 12 performs the transformation of $Z$ into $S$.

### Z-to-S. Version 1.0

**INPUT**

$Z=\{Z_1, Z_2, \ldots, Z_h\}$ where $Z_i$ is a tuple $<G_i, \text{ind}_i>$, $G_i \subseteq P$; $\text{ind}_i = \text{null}$ or $\text{ind}_i \in \{1, 2, \ldots, h\}$ and $\text{ind}_i \neq i$; $G_i \cap G_j = \emptyset$ for $i \neq j$. Besides, $G_1 \cup G_2 \ldots \cup G_h$ may not be equal to $P$. $\text{ind}_i$ is the index of the set that is parent of set $Z_i$. For example, if $Z_4$ is parent of $Z_5$, then $\text{ind}_5 = 4$. If $\text{ind}_i = \text{null}$ then $Z_i$ is a family root.

**OUTPUT:**

$S=\{S_1, S_2, \ldots, S_k\}$ where $S_i$ is a tuple $<H_i, K_i, \text{in}_x>$ such that $H_i \subseteq P$; $K_i = \emptyset$; $\text{in}_x = \text{null}$ or $\text{in}_x \in \{1, 2, \ldots, k\}$ and $\text{in}_x \neq i$; $H_i \cap H_j = \emptyset$ for $i \neq j$. Besides, $H_i = \emptyset$ if there exists $S_j$ such that $\text{in}_x = i$ (i.e., the set $H$ is empty in parent nodes). $\text{in}_x$ is the index of the parent subsystem of subsystem $S_i$.

1. $j = h$

2. FOR ($i=1; i \leq h; i++$) (* Traverse $Z$ *)

3. IF ($Z_i$ has children sets) THEN

4. $S_i = <\{ \}, \{ \}, \text{ind}_i>$

5. $j = j + 1$

6. $S_j = <G_i, \{ \}, i>$

7. ELSE

8. $S_i = <G_i, \{ \}, \text{ind}_i>$

9. ENDIF

10. END Z-to-S

Figure 12. Pseudocode of the Z-to-S algorithm
$S$ is required to produce a true hierarchical structure in which each internal node is formed by internal and leaf nodes. In addition, $S$ solves the following problem. Using the example in Figure 11, assume that file $f$ is used by 4 programs in group 5, and 4 programs in group 7. Clearly, $f$ cannot be assigned to any of the groups. It cannot be assigned to the parent group 2 either because programs in group 2 do not use this file. The solution to the problem is to create a parent node to which assign the file.

**assign-files. Version 1.1**

- Let $T(f, S)$ be the number of programs in $S$ that use file $f$. $S$ include the programs assigned to $S$ and the programs in all the nodes contained in the tree rooted in $S$.
- Let $\mathcal{F}(S)$ be the set of files used by programs in $S$. This set includes the files used by programs in all the nodes that form the tree rooted in $S$.

**INPUT**

(a) $S = \{S_1, S_2, \ldots, S_k\}$ where $S_i$ is a tuple $<G_i, K_i, ind_i>$ such that $G_i \subseteq P$; $K_i = \emptyset$; $ind_i =$ null or $ind_i \in \{1, 2, \ldots, k\}$ and $ind_i \neq i$; $G_i \cap G_j = \emptyset$ for $i \neq j$. Besides $G_i = \emptyset$ if there exists $S_j$ such that $ind_j = i$ (i.e., the set $G$ is empty in parent nodes). $ind_i$ is the index of the parent subsystem of subsystem $S_i$.

(b) Table $FT$ with 2 columns $<f, fq>$. $FT$ contains all files $f$ in the alpha-set ($f \in F$). $fq$ is the number of programs in $S$ that use file $f$. Initially, $fq = 0$. The algorithm calculates $fq$.

(c) $0.5 \leq \eta < 1$. $\eta$ is the minimum percentage of the total number of programs using a file $f$ that need to be in a group $G$ for $f$ to be assigned to $G$. Default value: $\eta = 0.5$.

**OUTPUT:**

$S$ having $K_i \subseteq F$ or $K_i = \emptyset$ such that $K_i \cap K_j = \emptyset$ for $i \neq j$. $K_i$ is the set of files assigned to subsystem $S_i$. However, $K_1 \cup K_2 \ldots \cup K_k$ may not be equal to $F$ because there are files that cannot be assigned to any subsystem.

```plaintext
1 FOR (i=1; i <= k; i++)
2    FOR EACH p \in G_i
3        FOR EACH f used by p increase fq in <f, fq> by one (FT[f],fq = FT[f],fq + 1 )
4 ENDFOR
5 FOR (i=1; i <= k; i++)
6    IF (ind_i = null ) THEN
7        FOR EACH f \in \mathcal{F}(S_j)
8            IF ((T(f,S_j) / FT[f],fq) \geq \eta ) THEN
9                allocate(f, i)
10 END assign-files

Figure 13. Pseudocode of the assign-files algorithm
```

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
Figure 13 contains the assign-files algorithm. The general idea of this algorithm is to find for each file the hierarchy tree (i.e., main subsystem) to which the file should belong, according to the file-assignment criterion. If none of the hierarchy trees satisfy the file-assignment criterion, the file becomes a common file. On the contrary, if a hierarchy tree X satisfies the file-assignment criterion, then the algorithm finds the node in the hierarchy X to which the file should be assigned. This process goes from the top to the lowers levels of X. In each level, all the nodes are tested against the file-assignment criterion. If a node satisfies the criterion, the file is assigned to that node. If no node satisfies the criterion, the algorithm goes to the next level of the hierarchy. This process is repeated until the right node is found.

Specifically, the algorithm starts by calculating for each file $f$ in the $S$ data structure the number of programs in $S$ that use $f$ (lines 1 to 4 in Figure 13). Then, for each hierarchy tree, the algorithm finds the files that can be assigned to that hierarchy (lines 5 to 10). Once the right hierarchy tree is located, the allocate function finds the right node in the hierarchy to assign the file (line 9).

The allocate function is shown in Figure 14. It is a recursive function. Assume that file $f$ is the file to be assigned to a particular node. In each level of recursion, a level of the hierarchy is checked to determine whether $f$ has to be assigned to that level. If allocate reaches a node with no children, then $f$ is assigned to this node (lines 1 to 4). If allocate reaches a node with more than one of its children using $f$, then $f$ is assigned to the node (lines 5 to 13). However, if allocates reaches a node with just one of its children using $f$, then allocate calls itself making that child the current node (line 14).

The output of the assign-files algorithm is a complete data cohesive subsystem decomposition of the subject system. Each subsystem is either a primitive subsystem or a complex subsystem.
3.5.3 The form-hierarchy algorithm

The next step of the ISA methodology is to use the form-hierarchy algorithm to form hierarchies of file implications. The input to this algorithm is the set of common files. The common files are the files that the assign-files algorithm could not assign to any subsystem. The output of the form-hierarchy algorithm is one or more trees that represent file implications as described in section 3.2.2. Figure 15 formally defines the input and output of this algorithm.

The form-hierarchy algorithm is shown in Figure 16. The algorithm has three major parts. First, candidate associations are selected from the N-List (lines 1 to 11). Next, the selected associations are sorted (line 12). Finally, the sorted associations are used to guide the construction of hierarchies of file implications of common files (lines 13 to 40).

The selection of candidate associations is based on two criteria. First, the two files in the association have to be common files (line 3). Second, the confidence of at least one of the files in the association has to be in the range $[\mu, 1]$ (i.e., $\mu \leq c \leq 1$) (lines 4 to 9). The parameter $\mu$
defines the lower possible confidence to allow the definition of a child-parent implication. Thus, a large value of \( \mu \) will lead to hierarchies of file implications with large confidences. Lines 5 and 8 add an association to the list of associations to be sorted in the second part of the algorithm (i.e., \( L \)). The idea is that \( L \) only contains associations whose first file has a confidence of at least \( \mu \). If both files in the association satisfy this constraint and their confidences are different, then two associations are added to \( L \); one in line 5 and the other in line 8. This mechanism is necessary to assure completeness in the sense that all implications are considered.

\[
\text{form-hierarchy. Version 1.1}
\]

**INPUT**

(a) \( CF = \{ f \in F \mid f \in K_i, i=1...k \} \). \( K_i \) are the file sets in the data structure \( S \). Thus, \( CF \) is the set of common files (i.e., \( f \) is not assigned to any subsystem).

(b) \( N\text{-List} = \{a_1, a_2, ..., a_m \} \) where \( a_i = s(f, g) \) is an association, \( s \) is the support of the association, and \( f, g \in F \). \( N\text{-List} \) contains 2-dimensional associations mined from AlphaN.

(c) \( 0.5 \leq \mu \leq 1 \). \( \mu \) is the minimum accepted confidence for defining a child-parent implication. Default value: \( \mu = 0.8 \).

(d) \( L = \{ \} \) is an empty list of associations.

**OUTPUT:**

The set of nodes that form file hierarchies \( R = \{ R_1, R_2, ..., R_r \} \) where \( R_i \) is a tuple \( < E_i, w_i > \) such that \( E_i \subset CF \) and \( E_i \neq \emptyset \); \( w_i = \text{null if } w_i \in \{ 1, 2, ..., r \} \) and \( w_i \neq i; E_i \cap E_j = \emptyset \) for \( i \neq j; E_1 \cup E_2 \cup ... \cup E_r = CF \); \( w_i \) is the index of the parent node of node \( R_i \). Nodes having \( w_i = \text{null} \) are hierarchy roots. Hierarchy roots with no children are called independent files.

**DEFINITIONS**

Let \( node(f) = R_i \mid f \in E_i \)

Let \( parent(f) = w_i \mid f \in E_i \)

Figure 15. Header of the form-hierarchy algorithm

The sorting of \( L \) is done in line 12. The sorting criteria are the confidence of the first file in the association (which is always at least \( \mu \)), the support, the product of the confidences of the two files, the id of the first file, and the id of the second file. These attributes are sorted in descending order, except the last two.
form-hierarchy

FOR (i=1; i ≤ m; i++)
    Let \( a_i = s[f, g] \) where \( a_i \in N\)-List
    IF \( f \in CF \text{ AND } g \in CF \) THEN
        IF \( c_f \geq \mu \) THEN (*\( c_f = s/ fq(f) \) *)
            \( L = L + s[f, g] \)
        ENDIF
        IF \( c_g \neq c_f \text{ AND } c_g \geq \mu \) THEN (*\( c_g = s/ fq(g) \) *)
            \( L = L + s[g, f] \)
        ENDIF
    ENDIF
ENDFOR

SORT(\( L, c_1(D), s(D), (c_1 \ast c_2)(D), f(A), g(A) \))
(*\( c_1 \) =confidence of the first file in association*)

FOR (i=1, j=0; i ≤ |\( L| ; i++ \)
    Let \( a_i = s[f, g] \) where \( a_i \in L \)
    IF (node\( (f) \) = null) THEN
        create new node \( R_{i+1} = < \{ f \}, \text{null}>; j = j+1 \)
    ENDIF
    IF (parent\( (f) \) = null) THEN
        IF (c_1 = c_2) THEN
            IF (node\( (g) \) = null) THEN
                Let \( R_v = node(f) \)
                \( E_v = E_v \cup \{ g \} \)
            ENDIF
            ELSE
                Let \( R_v = node(g) \)
                \( E_v = E_v \cup \{ f \} \)
                make children of node\( (f) \) be children of node\( (g) \)
                delete node\( (f) \)
            ENDIF-ELSE
        ELSE
            IF (c_f > c_g) THEN
                IF (node\( (g) \) = null) THEN
                    create new node \( R_{i+1} = < \{ g \}, \text{null}>; j = j+1 \)
                ENDIF
                Let \( R_v = node(f) \)
                Let \( R_v = node(g) \)
                \( w_v = s \)
            ENDIF
        ENDIF
    ENDIF-ELSE
END FOR

END form-hierarchy

Figure 16. Pseudocode of the form-hierarchy algorithm

76

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
These sorting criteria assure that the associations containing file implications with largest confidence are considered first. This order is important because the construction of the hierarchies of file implications is a bottom-up iterative process that builds the hierarchies based on the sorted associations, one at the time. Implications with high confidence form the first branches of the hierarchy tree. The rest of the criteria assures an order that is convenient to the construction of the hierarchy of file implications. In addition, the last two criteria were added to assure the order of the associations is the same regardless the sorting algorithm used to sort $L$. In other words, these criteria assure the repeatability of the process.

The construction of the hierarchies of file implications is an iterative process (lines 13 – 39). The general idea is to take one association at a time from $L$ and use it to define a child-parent relationship. This relationship may form a new hierarchy or be part of an existing one. A hierarchy of file implications is represented with a tree. Each node in the hierarchy tree contains a file. Nevertheless, when two or more files behave identically they go in a single node. Each node has a single parent and zero or more children. A node with no parent and no children contains an independent file. A node with no parent and one or more children corresponds to the root of a hierarchy of file implications.

According to the form of the associations in $L$, the first file implies the second file in the association. Thus, the second file goes in a parent node, and the first file goes in a child node. The first step to construct the hierarchy is to create a node for the first file in the implication (lines 15 and 16). Then, the algorithm checks if the first file in the association already has a parent. If so, the algorithm goes to the next iteration (line 17). This constraint along with the sorting criteria assure that every child-parent relationship in the hierarchies is the one having the largest confidence among all possible relationships involving the two files in the association under analysis. If the confidences of both files in the association are the same, then both files go in the same node (lines 18 to 27). If the confidences are not the same, then the
first file implies the second file and the first file goes in a node that is child of the node containing the second file (lines 28 to 37). The condition in line 29 avoids the creation of circular links in the hierarchy (i.e., that a parent node become a child of one of its children).

3.5.4 The subsys-merger algorithm

The objective of the subsys-merger algorithm is to produce a single hierarchy of subsystems. Up to this point in the ISA methodology, the subject system has been decomposed into several main subsystems. Each of these main subsystems is decomposed into a hierarchy of subsystems. We now need a single hierarchical tree to maintain consistency and to provide more information on how the main subsystems are related.

The underlying idea for this process is to organize the main subsystems into a single hierarchical tree in which each node is a main subsystem and the root node represents the complete system. The criterion used to create the hierarchy is file coverage. A main subsystem \( B \) is considered to be a child of main subsystem \( A \) if most of the files used by programs in \( B \) are used by programs in \( A \). Consequently, it is necessary to define a limit to the number of files in \( B \) that are not covered by \( A \). The \( \lambda \) parameter defines this limit. Of course, this parameter may prevent the main subsystems to be organized in a single hierarchy. Indeed, if this parameter is too restrictive, the subsys-merger algorithm may produce more than one hierarchy of main subsystems. This case suggests that the subject system contains two or more relatively independent subsystems.

The approach to this final merging is as follows. First, the main subsystems are sorted in ascending order according to the number of files used by all the programs in the main subsystems. Then, this order is used to find for each main subsystem \( A \) the main subsystems that cover it. Then, the main subsystem that "best" covers \( A \) is selected and assigned as the parent of \( A \). This process is repeated until all main subsystems are assigned to the hierarchy.
subsyp-merger 1.1

INPUT
(a) \( V = \{ S \in S \mid \text{ind} = \text{null} \} \) where \( S \) is a tuple \(<G, K, \text{ind}>\). \( V \) is the set of nodes representing family roots (root of a tree of subsystems).
(b) \( 0 < \lambda \leq 0.5 \). \( \lambda \) is the largest allowed percentage of files in a group that are not covered by its parent. Default value: \( \lambda = 0.3 \).

OUTPUT:
A list of the elements of \( V \) organized in several hierarchies. The format of the output is the following:
\[
S_a
S_b
S_c
S_d
S_e
S_f
S_g
\ldots
\]

subsyp-merger

1. \( \text{SORT}(V, |F(G_i)| \text{ (Ascending) }) \)
2. \( \text{FOR}(i = 1; i < |V|-1; i++) \)
3. \( \text{FOR}(j = i+1; j \leq |V|; j++) \)
4. \( \text{dist}[j] = |F(G_i) - F(G_j)| \)
5. \( \text{ENDFOR} \)
6. \( N = \{ l \mid \text{dist}[l] = \text{MIN}( \text{dist}[i+1], \ldots, \text{dist}[|V|] ) \} \)
7. \( \text{Let } u = l_1 \)
8. \( \text{IF } ( |F(G_i) - F(G_u)| / |F(G_i)| \leq \lambda ) \text{ THEN} \)
9. \( \text{Find } v : v \in N \wedge (|F(G_i) \cap F(G_v)| / |F(G_i)|) \times (|F(G_v) \cap F(G_i)| / |F(G_v)|) \text{ is maximum } \)
10. \( \text{Assign } G_v \text{ as the parent of } G_i \)
11. \( \text{ENDIF} \)
12. \( \text{ENDFOR} \)
13. \( \text{END subsyp-merger} \)

Figure 17. Pseudocode of the subsyp-merger algorithm

The criteria for “best” coverage have two components. First, the number of non-covered files has to be minimum. Second, the number of files shared by the child and the parent has to be maximum. For example, assume that main subsystem \( X \) uses 30 files, main subsystem \( Y \) uses 12 files, and main subsystem \( A \) uses 10 files. In addition, assume the files in \( X \cap A = 9 \)
and the files in $Y \cap A = 9$. Thus, $X$ and $Y$ cover 90% of the files in $A$. However, $Y$ is a better candidate to be the parent of $A$ because it is "closer." The closeness criterion is given by the product of two ratios. Each ratio is the ratio of common files between the parent and children subsystems and the total number of files in the parent and children subsystems. This concept is better explained using the numbers on the example given above. The product of the ratios of $Y$ and $A$ is $(9/12) * (9/10) = 0.675$, and the product for $X$ and $A$ is $(9/30) * (9/10) = 0.27$. Therefore, the product of the ratios of $Y$ and $A$ is larger. Thus, $Y$ and $A$ are "closer" to each other than are $X$ and $A$. Thereby $Y$ is selected as the parent of $A$. This mechanism may find later that $X$ is the "best" parent of $Y$. This mechanism maintains the right order in the hierarchy.

The subsys-merger algorithm is shown in Figure 17. The first line sorts the main subsystems. Then, lines 2 to 12 implement a loop in which a main subsystem is analyzed in each iteration. Assume that the algorithm is in iteration $i$, and that $A$ is the main subsystem being analyzed in the iteration. Lines 3 to 5 calculate for each main subsystem that follows $A$ in the sorting order (i.e., main subsystem with sorting index larger than $i$) the number of files in $A$ that they do not cover. Line 6 selects the main subsystems with maximum coverage of $A$. Line 8 verifies that the main subsystems that cover $A$ satisfy the required maximum percentage of non-covered files. There can be many main subsystems satisfying this constrain. Therefore line 9 selects the "best" (i.e., closest) main subsystem covering $A$, and line 10 assigns this main subsystem as the parent of $A$.

3.5.5 The TO-ISA representation

The final step of the ISA methodology is the presentation of the results. Two representation formats have been developed. The first representation format, called TO-ISA (Textual Output of ISA), is text based. The second format, called RM (Representation Model), is graph based. This section describes TO-ISA while chapter four describes RM.
The two main reasons to design TO-ISA are that it can be generated automatically and that it can be printed using any standard printer. Thus, the TO-ISA representation is easy to produce.

TO-ISA is divided in six sections to represent all the information produced by ISA. Each section contains a textual representation of different aspects of the hierarchical subsystem decomposition of the subject system. The name of the sections and their contents are the following.

i. HEADER SECTION. It contains general information of the analysis.

ii. COMMON FILES SECTION. It contains the details of the identified common files. It shows the independent files and the hierarchies of file implications.

iii. SUBSYSTEM SECTION. It contains the subsystem decomposition. For each subsystem, the programs that form the subsystem and the files assigned to it are listed.

iv. SINGULAR PROGRAMS SECTION. It lists the identified singular programs.

v. UNCONNECTED COMPONENTS SECTION. It lists the programs and files that are not included in the alpha set.

vi. SUPRA SUBSYSTEMS SECTION. It contains the results of the subsys-merger algorithm.

Table 4 lists all the products of the ISA methodology and the sections in TO-ISA where the products are presented. The remainder of this section describes each of the sections of TO-ISA. For this description, italics represent data items or variables, and regular-type words are fixed headers.

Figure 18 shows the format for the header section. It contains the name of the system, the processing date, the directories that contain the source code, the value of the parameters used to run the analysis, and the names of files and programs along with their ids. This list of
programs and files has to be sorted by id to facilitate the localization of the name of the programs and files because the rest of the sections use only the ids of programs and files.

Table 4. ISA products and their corresponding section in TO-ISA

<table>
<thead>
<tr>
<th>ISA Product</th>
<th>Section in TO-ISA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive subsystem</td>
<td>(iii) SUBSYSTEM SECTION</td>
</tr>
<tr>
<td>Complex subsystem</td>
<td>(iii) SUBSYSTEM SECTION</td>
</tr>
<tr>
<td>Main subsystem</td>
<td>(iii) SUBSYSTEM SECTION</td>
</tr>
<tr>
<td>Supra subsystem</td>
<td>(vi) SUPRA SUBSYSTEMS SECTION</td>
</tr>
<tr>
<td>Unconnected programs</td>
<td>(v) UNCONNECTED COMPONENTS SECTION</td>
</tr>
<tr>
<td>Singular programs</td>
<td>(iv) SINGULAR PROGRAMS SECTION</td>
</tr>
<tr>
<td>Unconnected files</td>
<td>(v) UNCONNECTED COMPONENTS SECTION</td>
</tr>
<tr>
<td>Common files</td>
<td>(ii) COMMON FILES SECTION</td>
</tr>
<tr>
<td>Independent files</td>
<td>(ii) COMMON FILES SECTION</td>
</tr>
<tr>
<td>Hierarchies of file implications</td>
<td>(ii) COMMON FILES SECTION</td>
</tr>
<tr>
<td>Link files</td>
<td>(iii) SUBSYSTEM SECTION</td>
</tr>
</tbody>
</table>

Figure 19 contains the format for the common files section. This section contains two subsections. The first subsection lists the independent files. For each independent file, the id is displayed as well as a list of the ids of all the subsystems that contain programs that use the independent file. The meaning of the notation is as follows: \textit{file-id} represents the file id of the independent file, \textit{sub-id} is the id of the subsystem containing programs using \textit{file-id}, and \textit{cardinality} is the number of programs in \textit{sub-id} that use \textit{file-id}. This notation may lead to overspecification of the list of subsystems using a particular common file. For example, assume that file \textit{fl3} is used by 5 programs in subsystem \textit{G}. Moreover, assume that \textit{G} is a child of \textit{F}, and \textit{F} a child of \textit{E}. According to the notation for this section and to the hierarchical organization of subsystems, the report would include the following line: \textit{fl3: E(5), F(5), G(5)}. Although the 5 programs in \textit{G} also belong to \textit{E}, this representation is redundant. To avoid this
overspecification, TO-ISA includes just the sub-id of the deepest subsystem in the hierarchy tree that uses file file-id. Thus, the correct representation of the example is fl3: G(5).

I. HEADER SECTION

(a) System Name
(b) Processing Date
(c) Analyzed directories
(d) Parameters:
   \[ \begin{align*}
   \beta &= 9 \\
   \gamma &= 9 \\
   \sigma &= 99 \\
   \delta &= 0.99 \\
   \tau &= 0.99 \\
   \varepsilon &= 0.99 \\
   \eta &= 0.99 \\
   \mu &= 0.99 \\
   \lambda &= 0.99
   \end{align*} \]
(e) Key
   - file-id: file-name
   - pgm-id: pgm-name

Figure 18. Format of the Header section

II. COMMON FILES SECTION

A. INDEPENDENT FILES
   - file-id: sub-id(cardinality), sub-id(cardinality), ...
   - file-id: sub-id(cardinality), sub-id(cardinality), ...

B. FILE HIERARCHIES
   FH-1:
   - file-id1: sub-id(cardinality), sub-id(cardinality), ...
   - file-id2, file-id3: sub-id(cardinality), sub-id(cardinality), ...
   - file-id4, ...: sub-id(cardinality), sub-id(cardinality), ...
   - file-id5, ...: sub-id(cardinality), sub-id(cardinality), ...
   - file-id6, ...: sub-id(cardinality), sub-id(cardinality), ...
   - ...
   FH-2:
   - file-id: sub-id(cardinality), sub-id(cardinality), ...
   - file-id, file-id: sub-id(cardinality), sub-id(cardinality), ...
   - file-id, ...: sub-id(cardinality), sub-id(cardinality), ...
   - file-id, ...: sub-id(cardinality), sub-id(cardinality), ...
   - ...

Figure 19. Format of the Common Files section

The second subsection in the Common Files Section describes the hierarchies of file implications. Each different hierarchy is assigned a unique identifier that is formed with the letters FH (for File Hierarchy) followed by a dash and a consecutive number. Thus, the file

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
hierarchies are labeled FH-1, FH-2, FH-3, and so on. The hierarchical relationships are represented with indentation. For example, in FH-1 in Figure 19, file-id1 is the root of the hierarchy. It has three children nodes, one contains file-id2 and file-id3, the second node contains file-id4, and the last one file-id5. file-id4 has also a child. The meaning of this hierarchy is that file-id5 implies file-id4 and file-id4 implies file-id1. Similarly, file-id2 implies file-id1. The notation to the right of each level of the hierarchy has a similar meaning as the notation used on the independent files subsection. However, in this subsection there is a distinction. In each line, a sub-id is included only in the list of the deepest file in the hierarchy that is used by sub-id. For example, consider the following definition:

FH-1:
A(10)
f5:
C(5)
f27:
fl3: D(3)
f6: E(7)
f39: K(8), P(6)

The meaning of this notation is that subsystem K has 8 programs that use files f39 and f5. Subsystem P has 6 files that use the same set of files. Subsystem E has 7 programs that use files f6 and f5. Subsystem D contains 3 programs that use files fl3, f27, and f5. There is no subsystem that only uses files f27 and f5. Subsystem C contains 5 programs that use only file f5. Finally, subsystem A has 10 programs that use all the files in the hierarchy. Although the programs in subsystem D use f5, this subsystem is not listed in the line for f5. D is only listed in the line for fl3. This mechanism simplifies the representation and takes advantage of the hierarchical relationships among files.

In addition, when a file is used by a singular program, the notation "program-id(*)" is used instead of "sub-id(cardinality)." This rule applies to both subsections.
Figure 20 shows the format for the Subsystem Section. As in the common files section, indentation is used to symbolize the different levels of the hierarchies of subsystems. In addition, each main subsystem is labeled with the letter S followed by a consecutive number. For example, the labels for main subsystems are S1, S2, S3, and so on. The label of an internal node in a given hierarchy is formed by using the label of its parent followed by a dot and a number. The number is used to differentiate the particular node from its sibling nodes. For example, if main subsystem S4 has 3 children nodes, they are labeled S4.1, S4.2, and S4.3.

III. SUBSYSTEM SECTION

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1.2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1.2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S1.2.3</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2.2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2.2.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 20. Format of the Subsystem section

As mentioned in section 3.2.2, each node in the hierarchy represents a subsystem. There are two types of subsystems, namely primitive subsystems and complex subsystems. One of the differences between them is that the former does not have children. In addition, complex subsystems do not have programs assigned to them. Consequently, there are two different notations for each type of subsystems. These notations are located to the right of the label of a subsystem as shown in Figure 20.
For a complex subsystem it is necessary to list the files assigned to it. The following notation is used to show these files:

\[
\{ \text{list-common-files} \}
\]

- \( \text{list-common-files} = (\text{file-id}_1 < \text{list-external-usage} >), (\text{file-id}_2 < \text{list-external-usage} >), \ldots \). This list can be empty. In this case, the subsystem does not have any file assigned to it.
- \( \text{file-id} \) is the id of a file that is assigned to the subsystem and is used by more than one of its child subsystems.
- \( \text{list-external-usage} = \text{target-id}_1, \text{target-id}_2, \ldots \). This list can be also empty.
- \( \text{target-id} \) is either a \text{pgm-id} or a \text{sub-id} outside the subsystem. Although \( \text{target-id} \) is assigned to another subsystem, it uses \( \text{file-id} \). In addition, if \( \text{target-id} \) is a subsystem, it is the deepest subsystem in its corresponding subsystem-tree where \( \text{file-id} \) is used. In short, if the \( \text{list-external-usage} \) list is not empty for a particular file \( \text{file-id} \), then this file is a link file.

A primitive subsystem contains more information. It has programs and files assigned to it. In addition, the programs assigned to a subsystem can be classified as core programs and non-core programs. Similarly, the files that the programs in a subsystem use can be either assigned to the subsystem or assigned to an external entity. External entities are other subsystems and hierarchies of file implications. The following notation is used to represent these features:

\[
(<l\text{-core-pgm-ids}>, l\text{-pgm-ids}), [ l\text{-internal-file-ids }], \{ l\text{-external-file-ids } \}
\]

- \( l\text{-core-pgm-ids} = \text{pgm-id}_1, \text{pgm-id}_2, \ldots \). In this list, \( \text{pgm-id} \) is the id of a program that is in the set of core programs of the subsystem. This is a non-empty list.
- \( l\text{-pgm-ids} = \text{pgm-id}_1, \text{pgm-id}_2, \ldots \). In this case, \( \text{pgm-id} \) is the id of a program assigned to the subsystem but not in the set of core programs. This list can be empty.

86
• $l\text{-}internal\text{-}file\text{-}ids = (file\text{-}id_1 < list\text{-}external\text{-}usage >), (file\text{-}id_2 < list\text{-}external\text{-}usage >),$ 
  
  ... . This list is similar to list\text{-}common\text{-}files, thereby list\text{-}external\text{-}usage = target\text{-}id_1,
  target\text{-}id_2, ... . file\text{-}id_i is a file assigned to the subsystem and target\text{-}id is a reference to
  an external entity that uses file\text{-}id.
• $l\text{-}external\text{-}file\text{-}ids = (file\text{-}id_1: entity\text{-}id_1), (file\text{-}id_2: entity\text{-}id_2),$ .... . In this list, file\text{-}id_i is
  the id of a file assigned to another subsystem but used by programs in the current
  subsystem, and entity\text{-}id_i is the id of the entity to which file\text{-}id_i is assigned. If file\text{-}id_i is
  a global common file, then SO is used as the entity\text{-}id. If file\text{-}id_i belongs to a file
  hierarchy, then the label of the hierarchy is used (e.g., $f6:FH\text{-}2$). Finally, if file\text{-}id_i is
  assigned to a subsystem, then the label of the subsystem is used. Either $l\text{-}internal\text{-}file\text{-}ids$
  or $l\text{-}external\text{-}file\text{-}ids$ can be empty, but not both.

### IV. SINGULAR PROGRAMS SECTION

<table>
<thead>
<tr>
<th>program-id (#files-used)</th>
<th>program-id (#files-used)</th>
<th>program-id (#files-used)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

### V. UNCONNECTED COMPONENTS SECTION

A. PROGRAMS

<table>
<thead>
<tr>
<th>program-id (#files-used)</th>
<th>program-id (#files-used)</th>
<th>program-id (#files-used)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

B. FILES

<table>
<thead>
<tr>
<th>file-id (#programs-using-file-id)</th>
<th>file-id (#programs-using-file-id)</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 21. Format of the Singular Programs and Unconnected Components sections

Sections IV and V of the TO-ISA report consist of a listing of programs and files. The
format for these sections is given in Figure 21. The Singular Programs Section lists the ids of
singular programs (i.e., program\text{-}id) and the numbers of files each singular program use (i.e.,
#files\text{-}used). A singular program cannot be assigned to any subsystem but is part of the alpha
set. The list is sorted by the number of files used by the program (descending order) and by
program id (ascending order). Thus, the singular programs that use more files are listed first.

87

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
VI. SUPRA SUBSYSTEMS SECTION

\[
\begin{align*}
    &main-subsys-1 \\
    &\quad main-subsys-2 \\
    &\quad main-subsys-3 \\
    &\quad main-subsys-4 \\
    &\quad main-subsys-5 \\
    &main-subsys-6 \\
    &main-subsys-7 \\
    &main-subsys-8
\end{align*}
\]

Figure 22. Format of the Supra Subsystems section

The Unconnected Components Section has two subsections. The first subsection lists the program ids of all the programs not contained in the alpha set. The list is similar to the list of programs in the Singular Programs Section. The sorting criterion is also the same. The second subsection lists the file ids of all the files not contained in the alpha set. Each entry in the list contains the file id of an unconnected file (i.e., \textit{file-id}) and the number of programs in the system using this file (i.e., \textit{#programs-using-file-id}). This list is sorted by \textit{#programs-using-file-id} in descending order and then by \textit{file-id} in ascending order.

VI. SUPRA SUBSYSTEMS SECTION

\[
\begin{align*}
    &S6 \\
    &\quad S1 \\
    &S3 \\
    &\quad S7 \\
    &\quad S5 \\
    &\quad S4 \\
    &\quad S2
\end{align*}
\]

Figure 23. Example of a Supra Subsystems section with more than one hierarchy

Finally, section VI presents the results of the subsys-merger algorithm. It presents the main subsystems organized hierarchically. Figure 22 shows the format of this section. In this figure, \textit{main-subsys-i} represents a main subsystem and the hierarchy is portrayed through indentation. The objective of the subsys-merger algorithm is to produce a single hierarchy to
represent the whole system. Depending on the value of the $\lambda$ parameter, the algorithm may produce more than one hierarchy. Therefore, the format in Figure 22 provides notation for this case. An example of a representation of a system that has more than one hierarchy is shown Figure 23.

In this chapter, we have described the ISA methodology. ISA uses data mining techniques to decompose a software system into a hierarchy of data cohesive subsystems. Specifically, ISA mines association rules from two sets of tuples, which describe the file-usage relationship among the programs and files in the subject system. A file-usage relationship between a program and a file is formed when the program reads or writes information in the file. ISA uses the mined associations to guide a bottom-up process that creates a hierarchy of data cohesive subsystems. First, ISA makes groups of programs that use similar data repositories (i.e., files). Then, ISA merges these groups to form larger groups. The merging is logical to allow the creation of a hierarchy of groups of programs. Next, ISA assigns files to the groups of programs. In this moment, the groups of programs become subsystems accordingly to the definition of subsystem adopted in this research. Next, ISA forms hierarchies of file implications with the files that were not assigned to any group of programs. ISA uses the mined associations to guide the construction of the hierarchy of file implications. Then, ISA organizes the larger subsystems into a single hierarchy that represents the complete system. Finally, ISA presents the results in two models, a text-based representation called TO-ISA, and a graph-based representation called RM. RM is described in chapter 4.

The definition of the ISA methodology was a feedback iterative process. In each iteration, current functionality was tested and new features were added. We used a reduced set of a real-world system to test and fine-tune ISA. Later, this reduced set was used as benchmark to test the tool that implements ISA.
The "junta" algorithm is basically an algorithm of the family of clustering algorithms based on centroids. The main idea of this type of algorithms is to define some centroids and then to use a distance function to group the points that are closer to each centroid. In junta, the concept of "core" of a group is similar to the concept of centroid, and the "is-member" and "is-subset" functions implement the distance function.

Finally, with regard to notation, the names alphaT and alphaN were derived from an initial stage of this research when we were defining how to build the set of tuples to feed the data mining algorithms. Our original attempt used tuples representing programs. This set of tuples can be represented as a table, where the rows represent programs and the columns files. However, later in the research we discovered that the transpose of this table (i.e., rows representing files and columns representing programs) was a better option for our purposes. Thus, the first table was regarded as the "Normal" alpha set, and the second table as the "Transpose" alpha set. Therefore, the name of the input sets to the data mining part of ISA are called alphaT (T meaning transposed set) and alphaN (N meaning normal set).
CHAPTER 4. THE REPRESENTATION MODEL

The Representation Model (RM) is a visual representation model that depicts graphically the results of the ISA methodology. The TO-ISA representation introduced in chapter 3 lacks the required expressiveness to highlight all the information produced by ISA. If the subject system is large, the resulting TO-ISA report is large, making it harder to identify and use the recovered design information.

Other options to represent the information produced by ISA include tree hierarchies, acyclic graphs, (k, 2)-partite graphs [Müll90], dendrograms [Lakh97], and higraphs [Hare88]. However, these representation models have similar problems as the TO-ISA representation when they are used to represent large systems. Also, none of them provides ways to represent all the information generated by ISA. Therefore, we define the graphical model RM that is capable of representing the design information produced by ISA. Our design choices were driven by the objective of producing a model that uses few building blocks and simple semantics to produce simple, yet expressive, diagrams. In addition, RM should represent the hierarchical subsystem decomposition. Finally, abstraction and the capability to represent large systems were also part of our design objectives.

4.1. Related Research

Lakhotia [Lakh97] considers that the problem of subsystem classification is essentially a graph partitioning problem. Thus, we can infer that graphs are a natural way to represent the components of a software system and their interrelationships. For instance, graphs are used to represent several aspects of software architectures [Chas96], and to represent references to global variables [Canf96].

Tree-based graphs are widely used to represent software subsystem decompositions because these decompositions can lead to hierarchical relationships among subsystems. For example, a tree is used in [Choi90] to represent a hierarchy of subsystems and modules. In
addition, some variations of trees that are used to represent hierarchies include dendrograms [Jard71] to represent layers of partitioned subsystem classifications [Lakh97], treemaps [John91] to visualize hierarchical information structures, and (k,2)-partite graphs [Müll90] to model multiple hierarchies.

An object model [Rumb91] is another representation of subsystem decomposition. Examples of research in this direction are [Snee95], [Pent96], and [Subr96]. Another approach for representing software artifacts uses browsers that combine text and graphs [Chan97].

RM benefited from several of these representation models. In addition, RM uses the underlying ideas of Venn Diagrams to represent subsystem membership. The use of layered diagrams to represent large systems, the notation for labeling each layer, and the use of abstraction levels are similar to the notation used for data flow diagrams. The representation of hierarchies of file implications resembles higraphs [Hare88]. Higraphs use rounded squares, called blobs, as the basic building block and RM uses rounded squares to represent subsets. Nested rounded squares represent set membership in both notations. In RM terms, set membership is used to represent levels in the hierarchy of file implications. In addition, higraphs use arcs to denote relationships among blobs, and RM uses arcs to represent file usage relationships.

4.2. Description of RM

RM uses graphical elements to represent the information produced by ISA. These elements are organized in diagrams. Each diagram contains the information of a particular subsystem. Accordingly, the diagrams are organized hierarchically to represent the hierarchical relationship among the subsystems identified by ISA. RM uses layered diagrams to represent the hierarchical structure of the subsystem decomposition. The top-layer diagram represents the top of the hierarchy. This diagram contains the main subsystems (i.e., complex

92

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
subsystems with no parent). Subsystems in lower levels of the hierarchy (i.e. complex subsystems and primitive subsystems) are represented with lower-layer diagrams.

4.2.1 Basic building blocks

Each of the products of the ISA methodology has a specific icon or graphic element to represent it. Table 5 shows the principal products of the ISA methodology and their corresponding RM notation.

Table 5. RM notation of the basic system components

<table>
<thead>
<tr>
<th>ISA product</th>
<th>RM representation</th>
<th>RM notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program</td>
<td>Square</td>
<td>pgm-id</td>
</tr>
<tr>
<td>File</td>
<td>Circle</td>
<td>file-id</td>
</tr>
<tr>
<td>Subsystem</td>
<td>Rectangle</td>
<td>Si</td>
</tr>
<tr>
<td>Link files set</td>
<td>Dotted rectangle</td>
<td></td>
</tr>
<tr>
<td>Singular programs set</td>
<td>Double-line rectangle</td>
<td></td>
</tr>
<tr>
<td>File-usage relationship</td>
<td>Directed arc</td>
<td></td>
</tr>
</tbody>
</table>

A square represents a program, and a circle represents a file. Inside the square and the circle are the program id and the file id, respectively. Both complex subsystems and primitive subsystems are represented with rectangles. The programs and files that belong to a subsystem are drawn inside the rectangle representing the subsystem. Each subsystem is labeled $S_i$, where $i$ is a consecutive integer starting at $i = 1$. The subsystem label is drawn in a small
square attached to any outer side of the rectangle representing the subsystem. Attached to the inner side of the rectangle is a dotted rectangle which contains the link files. Common files are drawn outside the subsystems, and singular programs are drawn inside a double-line rectangle.

Finally, it is necessary to define a way to express that a program or that all the programs in a subsystem use a file. We call this a file-usage relationship. A directed arc from a file to a subsystem or to a program represents a file-usage relationship. A solid arc from file $f$ to subsystem $Z$ means that every program and every sub-subsystem in $Z$ use $f$ while a dotted arc means that some programs or sub-subsystems in $Z$ use $f$. The file-usage relationships of singular programs are not represented graphically in RM because they introduce a great deal of unnecessary arcs to the diagram.

![Figure 24. RM notation for the hierarchies of file implications](image)

Figure 24 shows the notation to represent hierarchies of file implications as well as the semantics of the representation. The letters at the end of the arcs state the meaning of the corresponding file-usage relationship. RM uses a combination of implicit and explicit relationships to represent a hierarchy of file implications. The former uses location to express a relationship while the latter uses an arc. The hierarchy is represented implicitly using nested rounded squares. Files in deeper rounded squares correspond to files in the lower levels of the hierarchy. File-usage relationships are represented explicitly using arcs. The starting point of a file-usage relationship arc, noted by the dot, determines the files associated to it. Two

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
elements complete the notation to represent hierarchies of file implications. A small square attached to the outer side of the top-level rounded square represents all the files in the hierarchy and a small circle connects two or more file-usage relationships. Finally, a small square attached to the outer rounded square in the hierarchy contains the label of the hierarchy. The label notation is the same as in the TO-ISA representation: the name of a hierarchy is formed by the letters FH followed by a dash and a consecutive number $i$ such that $1 \leq i \leq \text{number of hierarchies}$ (e.g., FH-1, FH-2, ...). As Figure 24 shows, all possible file-usage relationships from a hierarchy of file implications can be represented by combining implicit and explicit relationships.

Figure 25 shows an example of a system level diagram in RM that uses the notation introduced above. Figure 25 represents a system that is decomposed into three main subsystems labeled S1, S2, and S3. Subsystem S1 contains four programs labeled 1, 2, 5, and
9, and two files labeled H and G. In this case, both files are also link files, therefore they are drawn inside a dotted rectangle. File H is used by all the programs in subsystem S2. This relationship is denoted with a solid arc (i.e., a complete file-usage relationship) that goes from the circle representing file H to the edge of the rectangle representing subsystem S2. In addition, file G is used by some of the programs in subsystem S3. This relationship is denoted with a dotted arc (i.e., a partial file-usage relationship) that goes from file G to the edge of the rectangle representing subsystem S3. Subsystems S2 and S3 are examples of subsystems that contain no files and one or more non-link files, respectively. Subsystem S3 contains file F. The notation means that this file is used by program inside S3 (i.e., programs 7, 8, 13, and 10) and by no other program in the system.

Figure 25 also shows three other major components of the system, a hierarchy of file implications, the set of singular programs, and an independent file. The hierarchy of file implications (FH-1) contains four files. The hierarchy contains two branches of implications: D → A (i.e., D implies A), and C → B → A. The arc that goes from the small box in the edge of the outer rounded square of the hierarchy of file implications to subsystem S1 means that all the programs in S1 use all the files in the hierarchy. The arc that goes from the box containing C to S2 means that all programs in S2 (i.e., programs 4, 14, and 17) use files C, B, and A.

Finally, there are two arcs that relate the hierarchy of file implications and subsystem S3. The dotted arc means that some of the programs in S3 use files D and A. The solid arc means that all the programs in S3 use file A. The singular programs are drawn inside a double-lined rectangle. In this case, the singular programs are programs 3, 6, and 15. Last, the figure includes one common file labeled E. File E is used by all the programs in subsystems S2 and S3.
4.2.2 Additional notation

RM contains additional notation. An hexagon represents a connector. A connector is used to identify the target of a file-usage relationship. One or more labels that provide the information of the destination of a file-usage relationship are inside the connector. A label has the form destination-subsystem-label/cardinality where the cardinality is the number of programs in the destination subsystem that use the file or set of files in the source part of the file-usage relationship arc. If no cardinality is given, then all the programs in the destination subsystem use the file(s). We defined the connector block to decrease the number of arcs in a diagram, to facilitate the diagram layout, and to reduce the number of overlapping arcs.

When a solid-line arc ends in the edge of a rectangle (i.e., subsystem), all programs inside the rectangle use the file(s) associated to the arc. However, some programs inside the rectangle may not use that particular file(s). Two notation elements denote this file-usage restriction. The label of the program or subsystem not using the file(s) is drawn between parenthesis over the file-usage relationship arc; and a triangle is drawn in the edge of the program or subsystem restricting the file-usage relationship.

Another semantic element is the cardinality of a partial file-usage relationship. The cardinality of a partial file-usage relationship is the number of programs in the target of the file-usage relationship arc that use the file in the source part of the arc. The cardinality is drawn over the arc representing the partial file-usage relationship. Another way to represent cardinality is with connectors. The reason to include the cardinality of a file-usage relationship arc is to provide an information element that can be used to determine the relative importance of a file-usage relationship. The lower the cardinality, the less critical the file-usage relationship because when the cardinality of a file-usage relationship is low it means that few programs are linked to the file(s) in the file-usage relationship.
Finally, when RM is used to represent very large systems, we draw sub-subsystems inside the main subsystem rectangles instead of programs. Similarly, we draw clusters of files instead of single files. The file clusters are represented with circles. The file cluster labeling follows the same rules as the labeling of subsystems, but the prefix letter is an F. For example, F1.2 is file cluster two in subsystem one. This procedure can be applied to inner layers until the subsystems and file clusters are simple enough to be represented in a single diagram.

4.2.3 Layers

RM uses layered diagrams along with a labeling mechanism to show the hierarchical subsystem structure. Layered diagrams facilitate the representation of large systems because layering permits representing the system at different abstraction levels. In the first layer, only major elements of the system are portrayed. More details are included in subsequent layers.

RM uses layered diagrams as follows. The first layer or top-layer contains the main subsystems, the common files among main subsystems, the hierarchies of file implications, and the cluster of singular programs. In the second layer, each main subsystem has a diagram that represents the main subsystem in greater detail. Similarly, each sub-subsystem drawn in a second layer diagram has a third-layer diagram that represent the sub-subsystem in greater detail. This layering continues until a primitive level is reached.

The labeling mechanism follows similar rules as the labeling of processes in data flow diagrams. Main subsystems are labeled with consecutive numbers starting at 1 and prefixing an S (i.e., S1, S2, ...). A low-level subsystem Z is labeled with its parent's label followed by a period and a consecutive number \( x = 1, 2, \ldots, n \), where \( n \) is the number of sub-subsystems belonging to the parent of Z (e.g., S1.2 identifies the sub-subsystem 2 in subsystem S1).

Figure 26 and Figure 27 show an example of the layering and labeling notation in RM diagrams. Figure 26 is the top layer diagram representing the entire system, and Figure 27 represents the S2 subsystem. The top layer diagram in Figure 26 shows the major components.
of the system. It contains 6 main subsystems, 2 hierarchies of file implications, one independent file, and the cluster of singular programs. The main subsystems are labeled S1 to S6 and are represented with rectangles. The files and programs assigned to each subsystem are drawn inside the rectangle representing the subsystem. An important observation is that the internal structure of each subsystem is not presented. In other words, the internal relationships within the subsystems are not shown.

Figure 26. Example of a system level RM diagram
The representation of the relationships within the subsystems is deferred to lower-level diagrams. The intention of this convention is to keep the diagram as simple as possible, thereby maintaining different levels of abstraction. In addition, some file-usage relationships are not drawn at this level to avoid overcrowding the diagram. For example, the file-usage relationships for the link files in S1 are not displayed. The diagram contains two hierarchies of file implications. The files in FH-1 play an important role in the system because all the subsystems access such files. The files in FH-2 are used by programs in subsystems S1, S4, S5, and by one program in S2. In addition, the figure shows that the system contains one common file, file 78. This file does not belong to any hierarchy of file implications but programs in four subsystems use it. The last main component is the cluster of singular programs. The system has 17 programs that cannot be included in any main subsystem. Figure
26 also provides some examples of connectors. For instance, a connector is used to indicate that 6 programs in S2 use file 48 in S5, and a connector with two labels is used to indicate the 4 programs in S5 and 1 program in S1 use independent file 78.

Figure 27 shows a second level diagram representing subsystem S2. The diagram represents S2 in more detail than the S2 representation in Figure 26, thereby the level of abstraction decreases. As the figure shows, all external entities (i.e., system components that does not belong to S2) are drawn outside the boundaries of S2. The external entities are the two hierarchies of file implications, files that are assigned to other subsystems but used by programs in S2, and programs assigned to other subsystems but using files in S2 (i.e., the link files in S2). In this layer, the internal structure of S2 is portrayed and some of the relationships within the S2 subsystem are represented. The subsystems belonging to S2 are represented, therefore the programs and files in S2 are drawn inside their respective subsystem. As in the top layer diagram, the internal structure of the subsystems in the diagram is not represented. Figure 27 also shows that some of the files assigned to S2 are internal common files. Files 30, 61, and 67 are examples of this type of files. Internal common files are used by most of the subsystems contained in a diagram. Therefore, they are drawn outside the rectangles representing the sub-subsystems but inside the outer rectangle representing the particular subsystem portrayed in the diagram. These internal common files are the files assigned to a complex subsystem.

A diagram in an inner layer representing a subsystem Z includes the "external" files used by the programs in Z, and the connections to the external programs or subsystems that access files inside Z. These external files and programs are located outside the subsystem boundaries. External files are drawn over a box that contains the label of the subsystem they belong to. This notation is used to represent external files 65, 66, 4, and 48 in the S2 diagram (Figure 27). The external programs or subsystems are represented with connector blocks and related to
the appropriate file with a file-usage relationship arc. For example, file 62 in S2.1 is used by 2 programs in subsystem S4. This relationship is represented by a file-usage relationship that goes from file 62 to a connector outside the boundaries of S2 (Figure 27). In addition, the diagram in Figure 27 exemplifies the incorporation of greater detail in lower-level layers. For instance, some of the file-usage relationships have been expanded: in the parent diagram (Figure 26) there is only one arc between FH-1 and S2, while in Figure 27 that arc has been replaced by 5 arcs. Some of these arcs go to the edge of S2 and others go to S2 subsystems.

Figure 27 also contains examples of the additional notation. For example, the arc that goes from external file 73 in FH-1 to the edge of S2 means that all programs inside S2 use files 73 and 75, except for program 15. To complete the file-usage restriction notation, program 15 in S2.3 includes a triangle to show that it is restricting this file-usage relationship. In addition, an example of a file-usage restriction involving a complete subsystem is shown by the annotation "(S2.4)" over the arc that goes from external file 76 in FH-1 to S2. It indicates that the programs in subsystem S2.4 do not use file 76. S2.4 has the corresponding triangle in its border line to indicate this file-usage restriction. Last, the figure also has some examples of cardinality. The arc that goes from external file 82 in FH-2 to S2.1 means that only one program in S2.1 uses files 82 and 39, and the connector that goes from internal file 61 to external subsystem S1 indicates that all the programs in sub-subsystem S1.1 use file 61.

In summary, an inner layer diagram follows the same rules as the top layer diagram. The main difference is that inner layer diagrams contain external entities while the top layer diagram does not.

4.3. Discussion

As with other diagramming models, layout is an issue for RM. From a positive perspective, there is only a list of programs and files inside the rectangles (sub-subsystems). In other words, the internal structure of the rectangles in a diagram is irrelevant. The structure is
defined in the next layer; thus, the only concern when drawing rectangles is to make them large enough to enclose the corresponding files and programs. Consequently, the layout problem is centered on the distribution of the rectangles and the hierarchies of file implications. This task is simplified by using connectors for file-usage relationships between distant entities. From a negative perspective, there is the problem of deciding which file-usage relationships are worth representing with an arc instead of with a connector. The driving decision factors are the representation of important relationships, the aesthetics of the diagram, and the elimination of overlapping arcs. Another situation is that the external files and programs may cause layout problems if many of them go to a single entity. Too many external entities require many file-usage relationship arcs which leads to overcrowded diagrams.

In addition to the layout problem, we identified the following limitations of RM. For clarity reasons, we do not include file-usage relationships that have low cardinality in the top layers of the diagram. This type of file-usage relationships introduces several arcs and connectors that are not relevant for the level of abstraction presented in the top layers. Nevertheless, these file-usage relationships are included in lower-level layers. The problem is determining the cardinality and the file-usage relationships that are “irrelevant” in each layer.

There is also an ambiguity problem when using file-usage relationships restrictions. For example, sub-subsystem S2.4 in Figure 27 has a file-usage relationship restriction. That is, programs in S2.4 do not use file 76. Since file 76 implies file 75, this file-usage restriction indicates that programs in S2.4 do not use file 75 either. However, there is a file-usage relationship that goes from file 63 to S2 indicating that all subsystems inside S2 use files 63 and 75. In short, the file-usage restriction indicates that file 75 is not used in S2.4 but the file-usage relationship indicates otherwise. This problem can be solved by adopting the convention that the file-usage restrictions are applied first and then the file-usage relationships. However, the current notation still may produce some ambiguity.
A low level diagram representing subsystem Z may contain many external files and many external programs using files in Z. Some of these external entities may be part of subsystems belonging to the same main subsystem as Z (a main subsystem is a subsystem in the top layer). This differentiation between external entities that belong to the same or to a different main subsystem is important because the former represent interfaces within the same main subsystem while the latter represent interfaces with other main subsystems. Although external files and programs belonging to the same main subsystem can be identified in a RM diagram using the entities' labels, RM does not provide a way to highlight this information.

As with other diagramming tools, RM has the advantage of offering a high-level visual representation of the subject system that facilitates its understanding. In addition, RM has other advantages. RM diagrams are capable of representing all the information produced by the ISA methodology. RM offers an easy-to-follow representation of the identified subsystems and their components. RM diagrams facilitate the detection of highly used files, subsystem interrelationships, and file relationships. In addition, RM makes easier the identification of the critical components of the system as well as the analysis of the system at several levels of abstraction. Therefore, RM can be used at several levels of a reverse engineering analysis or a maintenance effort. For example, the top-layer diagram provides a general view of the system that can be used by a project leader to form teams and to assign subsystems to each team. Similarly, each team leader can use lower layers of the diagrams to plan the analysis strategy, to detect important relationships with other teams, and to assign resources to the analysis of each program. Finally, programmers can use the RM diagrams to get the overall view of the system, to detect possible side effects, and to identify similar programs.

Another advantage of RM is its simplicity. It uses few building blocks and simple semantics. This feature implies that RM diagrams can be drawn with off-of-the-shelf CASE tools that support diagramming. Despite its simplicity, RM is capable of representing file
implication hierarchies, which increases the expressiveness of the diagrams. Therefore, RM can also be used to analyze and document transactional software systems that use large quantities of flat files or a relational database with many tables.
CHAPTER 5. EVALUATION OF ISA ON REAL-WORLD SYSTEMS

In chapter 1 we introduced the hypothesis that data mining is a feasible and valuable approach to reverse engineering and maintenance. In chapter 3 we introduced the ISA methodology to support the hypothesis. ISA decomposes a software system into a hierarchy of subsystems, which is a task performed in reverse engineering and maintenance. To close the description of this research, this chapter presents pragmatic results that provide evidence of the validity of the ISA methodology. In particular, this chapter shows practical results of applying ISA to two real-world systems, called the TRS system and the PS6 system.

The objective of this chapter is twofold. First, this chapter shows that ISA does produce a subsystem decomposition of the two subject systems. This pragmatic result provides positive evidence that the driving hypothesis of this research is correct. That is, that data mining is a valuable tool for reverse engineering and maintenance. Second, this chapter produces practical results that show the performance of the ISA methodology. The chapter includes several analyses of the subject systems using different values for the ISA parameters. These analyses help to understand the influence of each parameter in the final subsystem decomposition.

We used the RE-ISA tool to perform the subsystem decomposition of the two subject systems. RE-ISA (Reverse Engineering with ISA) is a software tool that automates the ISA methodology. The RE-ISA tool facilitated the application of ISA to the TRS and PS6 systems. This analysis was performed with the default values of the ISA parameters as defined in chapter 3. In addition, RE-ISA was applied to the TRS and PS6 systems using different values of the ISA parameters to understand the influence of each parameter in the final subsystem decomposition. The result of each experiment was compared to the decomposition obtained with the default parameter values, which was defined as the baseline for comparison. The experiments were separated in four categories.
(1) Experiments that vary the value of the parameters $\delta$, $\tau$, and $\varepsilon$ to identify their influence on the formation of groups of programs.

(2) Experiments that vary the value of the parameter $\eta$ to identify its influence on the assignment of files to groups of programs.

(3) Experiments that vary the value of the parameter $\mu$ to identify its influence on the formation of hierarchies of file implications.

(4) Experiments that vary the value of the parameter $\lambda$ to identify its influence on the final hierarchical arrangement of main subsystems.

Finally, this chapter proves an important feature of the ISA methodology: ISA can be automated. The RE-ISA tool is a completely automated tool that can produce the subsystem decomposition without any feedback from the user. The only information required by the tool is the location of the source code of the subject system. Nevertheless, RE-ISA also allows interaction with the user. For example, the user can modify the value of the ISA parameters, or repeat particular steps of the ISA methodology to fine-tune the final subsystem decomposition.

The rest of the chapter contains four sections. The first section provides a general description of the RE-ISA tool. The second and third sections describe the results of using ISA to decompose into subsystems the TRS and the PS6 systems, respectively. Moreover, these sections contain the results of the different experiments that we ran for each system. Finally, the fourth section includes a discussion of the results produced from these analyses.

5.1. Automation of ISA

One of the requirements for the ISA methodology is that it should be amenable to automation. We validated this requirement by defining the ISA algorithms at the pseudo-code level and then manually tracking each algorithm using a subset of the PS6 system. The results of this process confirmed both, that ISA produces a subsystem decomposition and that it can be automated. Nevertheless, the best way to validate the ISA automation requirement is by
building a software tool that automatically performs all the steps of the ISA methodology and by testing the tool with real-world systems. We developed the tool and named it RE-ISA.

Members of the Software Engineering Laboratory at Louisiana State University (LSU) assisted with the development of RE-ISA. The developers of RE-ISA were Aneesh Kulkarni and the author of this dissertation. Mr. Kulkarni participated in the architectural design and he did the detailed design and the programming of RE-ISA. The development of RE-ISA served as his Master of Science project, which is part of the requirements for the masters degree in System Science at LSU. The author of this dissertation defined the requirements and specifications of RE-ISA. In addition, he participated in the architectural design and the testing of the tool. The same subset of PS6 that was used to test ISA was used to test and fine-tune RE-ISA. The results of manually applying ISA to the subset of PS6 were used as the correctness benchmark for RE-ISA.

5.1.1 General description

The major features of RE-ISA are the following:

(1) RE-ISA is platform independent. RE-ISA consists of about 17,000 lines of code written in Java. RE-ISA can run on any computer system supporting this language without the need of recompilation. Indeed, RE-ISA was tested on three different environments: a Digital Alpha workstation running Digital Unix V4, a Sun Ultra Sparc workstation running Solaris, and a Pentium PC running Windows 95. In all cases, RE-ISA ran without any problem. To port RE-ISA to any of these computer systems, it is necessary to copy the RE-ISA Java object files to the system and set some global variables.

(2) RE-ISA uses a graphical user interface (GUI) to accept user input. RE-ISA uses windows and dialog boxes as the main interface with the user. The input to RE-ISA includes the location of the directory that contains the source code of the subject
system, the location of the directory that contains the “include” files of the subject system, the programming language on which the subject system was written (to determine the type of parser to use), the location of a working directory (to store temporary files and the results), and the values of the 9 ISA parameters.

(3) The output of RE-ISA is the textual representation TO-ISA as defined in section 3.5.5. RE-ISA presents TO-ISA in two forms: an ASCII file, and a screen window. The window shows just the part of the TO-ISA report that fits in the window according to the window size. Navigation buttons allow the user to see all the parts of TO-ISA. In addition, this window representation uses several text colors to highlight different parts of TO-ISA, which makes it easier to read the TO-ISA report.

(4) RE-ISA executes each one of the major steps of ISA independently. This feature means that RE-ISA does not need to execute all the steps of ISA at once. RE-ISA is built to allow multiple loops in the analysis of a particular system. That is, the user can repeat some steps without the need to repeat all the steps. When the user wants to repeat a particular step, all the steps that follow it also have to be repeated. For example, if the user already has analyzed a system but he or she wants to use a different set of parameters for group formation (step 7 of ISA), then steps 7 to 11 have to be performed again. RE-ISA implements this feature by storing in temporary files the output of each stage of the analysis. These outputs are the input of another stage. Therefore, when several steps need to be repeated, RE-ISA uses the output of the previous stage to re-start the analysis. RE-ISA defines the following stages: (i) Source code parsing (steps 1 and 2 of ISA), (ii) Alpha set generation (steps 3 and 4), (iii) Data mining (steps 5 and 6), (iv) Subsystem formation (steps 7 to 10), and (v) Generation of TO-ISA (step 11). Consequently, if the user wants to repeat the data mining stage using different parameters, he or she does not need to repeat the parsing.
nor the alpha set generation steps. This feature offers several advantages. The analyst can experiment with different parameter values because of the smaller turnaround time. Moreover, the analysis does not have to be done all at once. Furthermore, the results produced with different sets of parameters can be compared and the effect of a particular parameter can be singled out. Finally, this feature provides the required flexibility to use RE-ISA to analyze different types of systems. It is not necessary to develop a parser for every language. For example, if we want to use RE-ISA to analyze a system written in a proprietary database language with embedded SQL calls, we can use Unix tools and shell scripts to identify the programs and files in the subject system. Then, this information can be formatted to match the input to the stage ii of RE-ISA (i.e., Alpha set generation). Finally, RE-ISA can run starting on stage ii and the analyst can redo all subsequent stages as many times he or she wants. This scenario shows the flexibility of the RE-ISA tool in the sense that it can accept information generated with other tools. Furthermore, RE-ISA can be used as a clustering tool. RE-ISA can process any set of 2-dimensional associations.

(5) RE-ISA has a modular design and implementation. This feature means that any of the stages of RE-ISA can be substituted with another module that does the same but more efficiently. For example, when a better data mining algorithm is available, it can be plugged into RE-ISA without loss of functionality or side-effects. Furthermore, RE-ISA can include parsers for different systems. For example, if we want to use RE-ISA to analyze several systems written in a language X, we only need to develop a parser for that language and plug it into RE-ISA. The parser does not have to be a complete parser. It just needs the functionality to detect the different programs and files in the system.
RE-ISA can process large systems. The major concern in the construction of RE-ISA was its capability to process large amounts of information. Thus, running efficiency was not a major concern. Nevertheless, the response times are promising. For example, RE-ISA takes about a minute to process 220 K lines of COBOL code on a Digital Alpha workstation.

The following subsections discusses specific implementation details for each of the three main phases of the ISA methodology: generation of the database view, perform data mining, and consolidation and presentation of the results.

5.1.2 Generation of the database view

This phase of the ISA methodology contains 4 specific steps: (1) parse the source code, (2) assign unique identifications to programs and files, (3) generate the alpha set, and (4) build the alphaT and alphaN sets of tuples. RE-ISA implements these steps in two stages, each stage performing two steps.

The first stage uses a parser to identify the different programs and files that form the system, and to identify for each program the files it uses. Once the programs and files are identified, a unique identifier is assigned to each different program and file. The output of this stage is the ASCII files `pgm.txt`, `file.txt`, and `set.txt`. The `pgm.txt` file contains a list of all the programs in the system. For each program, the program name and its id are listed. The `file.txt` file is a similar list but for the files. The `set.txt` file contains a list of tuples of the form `<pgm-id, file-id>` where pgm-id is the identifier of a program and file-id is the identifier of a file. The meaning of the tuple is that the program pgm-id uses the file file-id. Thus, the `set.txt` file contains the "program uses file" relationship. The input to this stage is the location of the source code of the subject system, and the location of a temporary storage directory where RE-ISA stores the three output files of this stage. The current version of RE-ISA contains only
the parser for COBOL programs. It considers each source code file as a *program* and each data file defined with a “SELECT” statement as a *file*.

The second stage of RE-ISA takes the three files described above and produces the non-ASCII files *AlphaT.o* and *AlphaN.o*, which contain the alphaT and alphaN sets, respectively.

This design of RE-ISA allows that other parsers can be included in the tool. Besides, it allows the analysis of systems for which there is no parser available. For example, other tools can be used to produce the *pgm.txt*, *file.txt*, and *set.txt* files. After these files are stored in the temporary storage directory, the second stage of RE-ISA can be executed.

### 5.1.3 Data mining

This phase contains step 5 and 6 of ISA: (5) mine the alphaT set, and (6) mine the alphaN set. RE-ISA implements these two steps in a single stage. The output of this stage is the files *T_List.o* and *N_List.o*, which contain the mined 2-dimensional associations from the alphaT set and the alphaN set respectively. RE-ISA uses the Apriori algorithm [Agra94] to mine the 2-dimensional associations. Apriori mines association rules in two steps. First, Apriori finds all the item sets with transaction support greater than the threshold value minsup. That is, it finds all the sets of items contained in more than minsup transactions. These item sets are called large itemsets. Second, Apriori, generates the association rules based on the large itemsets found in the previous step (This process is explained in section 2.1).

During the first step, Apriori does several passes over the data set. In pass k, Apriori finds large itemsets of size k called k-itemsets. Specifically, for each iteration k, Apriori generates candidate sets using the (k-1)-itemsets and then traverse the tuples to calculate the support for each candidate set. The candidate sets with support > minsup form the k-itemset. The process continues until Apriori cannot generate more candidate sets.

RE-ISA only uses the first step of Apriori because the ISA methodology requires only 2-dimensional associations (i.e., large itemsets of size 2). Nevertheless, since RE-ISA
implements completely the first step of the Apriori algorithm, Re-ISA can mine n-dimensional associations. One of the RE-ISA input screens asks for the maximum dimension of associations that the user wants to be mined. The default value is 2.

5.1.4 Consolidation and presentation of the results

This last phase of the ISA methodology contains steps 7 to 11. Steps 7 to 10 consist of applying specific algorithms to produce the subsystem decomposition, and step 11 is devoted to the presentation of the results. In step 7, the junta algorithm is used to form groups of programs. In step 8, the assign-files algorithm assigns files to particular groups of programs (subsystems). In step 9, the form-hierarchy algorithm forms hierarchies of file implications. Finally, in step 10 the subsys-merger algorithm is used to form the final hierarchical arrangement of main subsystems. RE-ISA implements steps 7 to 10 in a single stage. The input to this stage is the $T_{List.o}$ and $N_{List.o}$ files produced in the previous stage. The output is the $group.o$ file, which contains all the subsystem decomposition information. The algorithms in this stage use 6 parameters. RE-ISA uses the default parameter values defined in chapter 3 but it allows the used to modify the parameter values.

The last stage of RE-ISA is the implementation of step 11. RE-ISA takes the group.o file as input and generates two versions of the TO-ISA report: a file version and a screen version. The file version is an ASCII file that contains the TO-ISA report as defined in section 3.5.5. The screen version is the same as the file version but it is presented on the screen, and it uses colored text to differentiate the components of the TO-ISA report.

Following the design philosophy of the RE-ISA tool, RE-ISA is prepared to receive the module that will implement the graphical representation of the results (i.e., the RM model described in chapter 4). The idea is that the RM graphical representation will be an independent stage of RE-ISA. The input to this new stage will be the group.o file and the output will be a series of RM diagrams displayed on screen.
This general overview of RE-ISA described the major parts and capabilities of the tool. The next two sections describe our experiences with the application of RE-ISA to analyze two COBOL systems.

5.2. Case Study: The TRS System

The first case study is a 14 year old system called the Teachers Retirement System (TRS). TRS, which runs on an IBM-370, consists of approximately 25,000 lines of COBOL code distributed into 28 source code files. For the analysis of TRS, each source code file is considered an independent program, and each data repository defined by a "SELECT" statement is considered a file.

First, we ran RE-ISA using the default value of all the parameters to decompose TRS into subsystems. Then, we ran RE-ISA several times varying the value of one parameter at a time to observe how it affected the final subsystem decomposition.

The first run used the default value of the ISA parameters. The parameters and their default value are: $\beta = 1, \gamma = 1, \sigma = 2, \delta = 0.5, \tau = 0.3, \varepsilon = 0.4, \eta = 0.5, \mu = 0.8,$ and $\lambda = 0.3$ (chapter 3 describes these parameters). The complete analysis from parsing the source code to the merging of main subsystems was done in 11.97 seconds running RE-ISA on a Digital Alpha Workstation. The textual output of the results (i.e., TO-ISA representation) is shown in Figure 28 to Figure 31.

Figure 28 contains the header section. It shows that the TRS system contains 28 programs and 38 files. Most of the programs use few files. For example, 19 programs (67%) use 3 or less files. In addition, three programs use many files. Programs p18 and p19 use 18 files each, and p16 uses 17 files. The remaining 6 programs use between 4 and 6 files. Also, most of the files (89%) are used by three programs or less. Only 4 files are used by many programs. File f2 is used by 22 programs, file f1 is used by 20 programs, and files f3 and f17 are used by 9 programs each. These numbers suggest that this system has a monolithic structure in which
few programs do most of the functions and few files store most of the information. It is likely that files that are used only by one or two programs are temporary files.

### I. HEADER SECTION

(a) SYSTEM NAME: TRS  
(b) PROCESSING DATE: 18 Jan 1999 18:11:15 GMT  
(d) PARAMETERS  
\[
\begin{align*}
\text{alpha} & = 1.0 \quad \text{beta} = 1.0 \quad \text{support} = 2.0 \quad \text{delta} = 0.5 \quad \text{tau} = 0.3 \\
\text{epsilon} & = 0.4 \quad \text{eta} = 0.5 \quad \text{mu} = 0.8 \quad \text{lambda} = 0.3
\end{align*}
\]

(e) KEY – PROGRAMS

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th># of files</th>
<th>ID</th>
<th>Name</th>
<th># of files</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>trs1002</td>
<td>(2)</td>
<td>p2</td>
<td>trs1004</td>
<td>(2)</td>
</tr>
<tr>
<td>p3</td>
<td>trs1006</td>
<td>(1)</td>
<td>p4</td>
<td>trs1008</td>
<td>(1)</td>
</tr>
<tr>
<td>p5</td>
<td>trs1010</td>
<td>(3)</td>
<td>p6</td>
<td>trs1012</td>
<td>(3)</td>
</tr>
<tr>
<td>p7</td>
<td>trs1013</td>
<td>(4)</td>
<td>p8</td>
<td>trs1014</td>
<td>(3)</td>
</tr>
<tr>
<td>p9</td>
<td>trs1012</td>
<td>(2)</td>
<td>p10</td>
<td>trs1014</td>
<td>(2)</td>
</tr>
<tr>
<td>p11</td>
<td>trs1016</td>
<td>(1)</td>
<td>p12</td>
<td>trs1108</td>
<td>(1)</td>
</tr>
<tr>
<td>p13</td>
<td>trs1110</td>
<td>(3)</td>
<td>p14</td>
<td>trs1112</td>
<td>(3)</td>
</tr>
<tr>
<td>p15</td>
<td>trs1114</td>
<td>(3)</td>
<td>p16</td>
<td>trs3002</td>
<td>(17)</td>
</tr>
<tr>
<td>p17</td>
<td>trs3004</td>
<td>(3)</td>
<td>p18</td>
<td>trs3006</td>
<td>(18)</td>
</tr>
<tr>
<td>p19</td>
<td>trs3007</td>
<td>(18)</td>
<td>p20</td>
<td>trs3008</td>
<td>(5)</td>
</tr>
<tr>
<td>p21</td>
<td>trs3010</td>
<td>(2)</td>
<td>p22</td>
<td>trs3012</td>
<td>(2)</td>
</tr>
<tr>
<td>p23</td>
<td>trs3014</td>
<td>(6)</td>
<td>p24</td>
<td>trs3016</td>
<td>(3)</td>
</tr>
<tr>
<td>p25</td>
<td>trs3020</td>
<td>(4)</td>
<td>p26</td>
<td>trs3022</td>
<td>(6)</td>
</tr>
<tr>
<td>p27</td>
<td>trs3024</td>
<td>(3)</td>
<td>p28</td>
<td>trs3032</td>
<td>(6)</td>
</tr>
</tbody>
</table>

(e) KEY – FILES

<table>
<thead>
<tr>
<th>ID</th>
<th>File Name</th>
<th># of pgms</th>
<th>ID</th>
<th>File Name</th>
<th># of pgms</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>PPM-FILE</td>
<td>(20)</td>
<td>f2</td>
<td>TRS-FILE</td>
<td>(22)</td>
</tr>
<tr>
<td>f3</td>
<td>PRINT-FILE</td>
<td>(9)</td>
<td>f4</td>
<td>TRS2-FILE</td>
<td>(1)</td>
</tr>
<tr>
<td>f5</td>
<td>REPORT-FILE</td>
<td>(2)</td>
<td>f6</td>
<td>SYSIN1-FILE</td>
<td>(3)</td>
</tr>
<tr>
<td>f7</td>
<td>CHECK-FILE</td>
<td>(3)</td>
<td>f8</td>
<td>RAS-NEG-SUPP-FILE</td>
<td>(3)</td>
</tr>
<tr>
<td>f9</td>
<td>WORK-FILE-1</td>
<td>(3)</td>
<td>f10</td>
<td>WORK-FILE-2</td>
<td>(3)</td>
</tr>
<tr>
<td>f11</td>
<td>WORK-FILE-3</td>
<td>(3)</td>
<td>f12</td>
<td>WORK-FILE-4</td>
<td>(3)</td>
</tr>
<tr>
<td>f13</td>
<td>NEGATIVE-CHK-FILE</td>
<td>(3)</td>
<td>f14</td>
<td>PPM-DUMP-FILE</td>
<td>(3)</td>
</tr>
<tr>
<td>f15</td>
<td>CHK-DUMP-FILE</td>
<td>(3)</td>
<td>f16</td>
<td>STATE-TAPE-FILE</td>
<td>(1)</td>
</tr>
<tr>
<td>f17</td>
<td>SORT-FILE</td>
<td>(9)</td>
<td>f18</td>
<td>STATE-PRINT-FILE</td>
<td>(3)</td>
</tr>
<tr>
<td>f19</td>
<td>LSU-PRINT-FILE</td>
<td>(3)</td>
<td>f20</td>
<td>ERROR-FILE</td>
<td>(3)</td>
</tr>
<tr>
<td>f21</td>
<td>TEACH-DISK-FILE</td>
<td>(1)</td>
<td>f22</td>
<td>TEACH-TAPE-FILE</td>
<td>(1)</td>
</tr>
<tr>
<td>f23</td>
<td>SYSIN2-FILE</td>
<td>(2)</td>
<td>f24</td>
<td>ORP-TAPE-FILE</td>
<td>(2)</td>
</tr>
<tr>
<td>f25</td>
<td>TRS-DISK-FILE</td>
<td>(1)</td>
<td>f26</td>
<td>TRS-TAPE-FILE-IN</td>
<td>(1)</td>
</tr>
<tr>
<td>f27</td>
<td>WORK-FILE</td>
<td>(1)</td>
<td>f28</td>
<td>TRS-TAPE-FILE-OUT</td>
<td>(1)</td>
</tr>
<tr>
<td>f29</td>
<td>TRS-TAPE-FILE</td>
<td>(1)</td>
<td>f30</td>
<td>TRS-VSAM-FILE</td>
<td>(2)</td>
</tr>
<tr>
<td>f31</td>
<td>TRS-BACKUP-FILE</td>
<td>(1)</td>
<td>f32</td>
<td>CORR-REPORT-FILE</td>
<td>(2)</td>
</tr>
<tr>
<td>f33</td>
<td>CORR-REPORT-FILE-D</td>
<td>(2)</td>
<td>f34</td>
<td>ERROR-REPORT-FILE</td>
<td>(2)</td>
</tr>
<tr>
<td>f35</td>
<td>CARD-FILE</td>
<td>(1)</td>
<td>f36</td>
<td>TRSL-TAPE-FILE</td>
<td>(1)</td>
</tr>
<tr>
<td>f37</td>
<td>PRINT-FILE-1</td>
<td>(1)</td>
<td>f38</td>
<td>PRINT-FILE-2</td>
<td>(1)</td>
</tr>
</tbody>
</table>

Figure 28. Header section of the TO-ISA report for the TRS system
Figure 29. Sections IV, V, and VI of the TO-ISA report for the TRS system

Figure 29, Figure 30, and Figure 31 show different aspects of the subsystem decomposition. ISA organized 20 programs (71%) in 5 main subsystems. Only 8 programs were not assigned to any subsystem (unconnected programs in Figure 29). The unconnected programs are those that were not included in the alpha set. Four of these programs use only one file, therefore they did not satisfy the minimum support ($\sigma = 2$) required to enter the alpha set. The other 4 programs use two or more files. However, they were left out the alpha set because they use files that are used by less than two programs. For a detailed explanation of this behavior, refer to the algorithm that creates the alpha set (chapter 3). From the 38 files in the system, one file (f17) is an independent file, 22 files (57%) are assigned to subsystems, and 15 (39%) are unconnected files. Figure 29 shows that most of the unconnected files are used by only one program. These files were not included in the alpha set.
Figure 30 shows that f17 is the only common file. Programs in three subsystems use f17: 3 programs in subsystem S1.1, 3 programs in S2.1 and 2 programs in S5.1. Since there is only one common file, no hierarchies of file implication were produced.

# II. COMMON_FILES SECTION

# A. INDEPENDENT FILES

f17 : S1.1(3) S2.1(3) S5.1(2)

# B. FILE HIERARCHIES

Figure 30. Common files section of the TO-ISA report for the TRS system

Figure 31 shows the subsystem decomposition. ISA identified 5 main subsystems. Subsystem S1 contains 3 programs (i.e., p18, p19, and p16) and 15 files (i.e., f6, f7, f8, f9, f10, f11, f12, f13, f14, f15, f18, f19, f20, f23, and f24). Programs in S1 use three external files: files f1 and f2 that belong to subsystem S3, and f17 that is a common file. Subsystem S2 has 3 programs and 3 files. Programs in this subsystem also use f1, f2, and f7. Subsystem S3 contains two sub-subsystems, and two files (f1 and f2) that are used by programs in both subsystems. Subsystem S3.1 contains 6 programs and one file, and S3.2 contains 4 programs and no files. Programs in S3 do not use any external files. Thus, S3 can be regarded as an "independent" subsystem because it does not use any external data repository. In other words, programs in S3 only access files that belong to S3. Finally, subsystems S4 and S5 contain two programs each. Subsystem S4 has one file and S5 has no files. These subsystems have programs that access files in S3 and the independent file f17.

One interesting result of this analysis is that programs in all subsystems use files f1 and f2, yet they are assigned to S3 instead of being classified as common files. The reason for this behavior is that $\eta = 0.5$. Recall that $\eta$ defines the minimum percentage of programs using a file that have to be in a subsystem $X$ so that the file is assigned to $X$. In this case, file f1 is
used by 20 programs and S3 contains 50% of them. Thus, f1 is assigned to S3 because \( \eta = 0.5 \) (or 50%) and S3 satisfies this condition.

```
# III. SUBSYSTEMS SECTION

S1  { }
S1.1 (<p18 p19 > p16 )
    [ (f6< >) (f7< >) (f8< >) (f9< >) (f10< >) (f11< >) ]
    [ (f12< >) (f13< >) (f14< >) (f15< >) (f18< >) ]
    [ (f19< >) (f20< >) (f23< >) (f24< >) ]
    [ (f1 : S3 ) (f2 : S3 ) (f17 : S0 ) ]

S2  { }
S2.1 (<p23 p28 > p26 )
    [ (f2< >) (f3< >) (f4< >) ]
    [ (f1 : S3 ) (f2 : S3 ) (f17 : S0 ) ]

S3  { (f1< S1.1 S2.1 S3.1 S3.2 S4.1 S5.1 >)
     (f2< S1.1 S2.1 S3.1 S3.2 S4.1 >) }
S3.1 (<p5 p6 p8 p13 p14 p15 >)
     [ (f3< S5.1 >) ]
     [ (f1 : S3 ) (f2 : S3 ) ]

S3.2 (<p1 p2 p9 p10 >)
     [ ]
     [ (f1 : S3 ) (f2 : S3 ) ]

S4  { }
S4.1 (<p7 p24 >)
     [ (f5< >) ]
     [ (f1 : S3 ) (f2 : S3 ) ]

S5  { }
S5.1 (<p25 p27 >)
     [ ]
     [ (f1 : S3 ) (f3 : S3.1 ) (f17 : S0 ) ]
```

Figure 31. Subsystem section of the TO-ISA report for the TRS system

Finally, section VI in Figure 29 shows that the merger of main subsystems (step 10 of ISA) produced no hierarchical organization of main subsystems. This result suggests that the
subsystems are relatively independent. Indeed, only three files are shared by most of the subsystems (i.e., files f1, f2, and f17).

Figure 32. First-layer RM diagram of the TRS system

Figure 32 and Figure 33 show the graphical representation of this decomposition using the RM model. The diagram in Figure 32 represents the top layer of the hierarchy. It shows the five main subsystems and the programs and files that are contained in them. The diagram stresses the facts that S3 is an independent subsystem and that files f1, f2, and f17 play a predominant role in the system. One interesting result highlighted by the diagram is that S5 is the only external subsystem that uses f3. A logical decision would be to make S5 a subsystem of S3. However, S3 would have to include a link to f17, and it would lose its internal data cohesiveness.
This example shows that ISA promotes subsystem decompositions that maintain internal data cohesiveness. Figure 33 depicts the S3 subsystem in detail. It shows that the programs inside S3 are organized in two sub-subsystems, that f1 and f2 are common to both subsystems, and that f3 belongs to one of the sub-subsystems. The diagram shows the major relationships of S3 with other subsystems and that S3 does not use any external file. This representation of the TRS system allows the rapid identification of critical resources, the major relationships among subsystems, and the overall organization of the system. These diagrams demonstrate that the RM model facilitates the understanding and visualization of the subsystem decomposition.

Figure 33. RM diagram of the S3 subsystem

This part of the case study shows that the ISA methodology decomposes the TRS system into subsystems. In addition, it shows some of the advantages of the methodology; ISA can be completely automated, ISA can produce a data cohesive subsystem decomposition without any previous knowledge of the subject system, and the RM model can represent the results produced by ISA and facilitates the visualization of such results.
The next part of the case study consists on varying the value of the parameters to determine how each parameter influences the result. To this end, several experiments were run. In each experiment, one parameter is selected and assigned a particular value. The rest of the parameters are set to their default value. Then, the TRS system is decomposed in subsystems using that particular set of parameter values. The resulting subsystem decomposition is compared to the decomposition produced with the default parameter values (i.e., the result shown in Figure 28 to Figure 31). We call this result the default subsystem decomposition. For these experiments, three parameters maintain their default value in all the experiments. These parameters are \( \beta \), \( \gamma \), and \( \sigma \). The reason for this decision is that these parameters control the programs and files that go into the alpha set, and the minimum support for the associations. If these parameters were increased, then the number of programs in the alpha set would be reduced because the TRS system contains many programs that use few files and many files that are used by few programs. Thus, if only few programs and files enter the alpha set, then the subsystem decomposition will become trivial (i.e., one subsystem containing all programs and files).

To facilitate the understanding of the effect of each parameter in the result, we include Table 6. The table shows the programs and files in the TRS system grouped according to the result produced with the default parameter values. The table has the programs in rows and the files in columns. The intersection of a row with a column is marked if the program represented by the row uses the file represented by the column. The programs that belong to the same subsystem are in adjacent rows, and the unconnected programs are listed in the last 8 rows.

The experiments are divided in four sets. Experiments in the first set target the parameters that influence the formation of groups of programs (i.e., \( \delta \), \( \tau \), and \( \varepsilon \)). Experiments in the second set target the parameter \( \eta \), which controls the assignment of files to groups of programs. Experiments in the third and fourth sets target the parameters that control the
formation of hierarchies of file implications ($\mu$) and the final hierarchical arrangement of main subsystems ($\lambda$), respectively.

Table 6. Programs and files in the TRS system

|    | F1 | F2 | F3 | F4 | F5 | F6 | F7 | F8 | F9 | F10 | F11 | F12 | F13 | F14 | F15 | F16 | F17 | F18 | F19 | F20 | F21 | F22 | F23 | F24 | F25 | F26 | F27 | F28 | F29 | F30 | F31 | Tot |
|----|----|----|----|----|----|----|----|----|----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|----|
| S1 | X  | X  | X  | X  | X  | X  | X  | X  | X  | X   | X   | X   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| S2 | X  | X  | X  | X  | X  | X  | X  | X  | X  | X   | X   | X   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| S3.1| X  | X  | X  | X  | X  | X  | X  | X  | X  | X   | X   | X   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| S3.2| X  | X  | X  | X  | X  | X  | X  | X  | X  | X   | X   | X   |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| S4 | X  | X  | X  | X  | X  | X  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| S5 | X  | X  | X  | X  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| S6 | X  | X  |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |
| S7 |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |     |

The first experiments target $\delta$. Its default value is 0.5 and its range is $0 < \delta < 1$. This parameter restricts the associations considered in the analysis. That is, the product of the confidences of the elements in an association $a$ has to be equal or larger than $\delta$ in order for $a$ to be included in the analysis. Thus, as $\delta$ becomes larger fewer associations enter the analysis. We ran three experiments using $\delta = 0.7$, $\delta = 0.85$, and $\delta = 0.95$. The expected result was that as $\delta$ becomes larger fewer programs become part of the subsystems. Thus, fewer groups of programs and groups with fewer programs would be produced. However, in the three experiments, we obtained the same results as the default subsystem decomposition. This behavior derives from the fact that the programs in the same group use almost the same set of files (see Table 6). Thus, the confidences of the associations are high. In this case, this parameter has no effect.
The second round of experiments targets $\tau$ (default value: $\tau = 0.3$; range: $0 < \tau \leq 0.5$). This parameter controls whether a program is assigned to a certain group of programs. The larger the value of $\tau$, the easier for a program to enter a group. The expected behavior is that when $\tau$ is large, the groups would contain more programs or some programs would go to different groups. We ran two experiments using $\tau = 0.1$ and $\tau = 0.5$. In the first case, we obtained the default subsystem decomposition. In the second case, there was one change. Program p26 was assigned to S3.2 instead of to S2.1. As Table 6 shows, the reason for this result is that p26 covers all the programs in S3.2 (see chapter 3 for the definition of "coverage"). When $\tau$ is more restrictive, p26 goes to S2.1 because the confidence between p26 and programs in S2.1 is larger that the confidence between p26 and programs in S3.2.

The last round of the first set of experiments targets $\epsilon$ (default value $\epsilon = 0.4$; range $0 < \epsilon \leq 0.5$). This parameter performs a function similar to the function performed by $\tau$. In particular, $\epsilon$ controls the number of files allowed to be used only by one program in a group of programs. Thus, $\epsilon$ assures that a program $p$ does not enter the group $G$ if $p$ use many files that are not used by the programs already in $G$. As $\epsilon$ becomes larger, the constraint that it controls weakens. Therefore, it is expected that with large values of $\epsilon$, the groups may contain more programs and the number of different groups may be reduced. We ran two experiments using $\epsilon = 0.5$ and $\epsilon = 0.1$. In both cases, the result was the same as the result of the default subsystem decomposition. The reason for this behavior is that the programs inside a group use mostly the same set of files. No new files are introduced to the group when a new program is added to the group. Therefore, for this specific system, $\epsilon$ represents no constraint.

The second set of experiments focused on varying the value of $\eta$ (default value $\eta = 0.5$; range $0.5 \leq \eta < 1$). Recall, $\eta$ controls the assignment of files to particular groups of programs. The default value of $\eta$ means that in order to assign file $f$ to subsystem $X$, subsystem $X$ should
contain 50% or more of the programs that use \( f \). Thus, as \( \eta \) becomes larger, it is expected that fewer files would be assigned to subsystems. In other words, it is expected an increment in the number of common files. This increment also increases the chances of formation of hierarchies of file implications. We ran two experiments using \( \eta = 0.7 \) and \( \eta = 0.9 \). The results were as expected. In the first case, files \( f_1 \) and \( f_2 \) went out of \( S_3 \) and became common files. Moreover, a hierarchy of file implications was formed, and it included the other common file \((f_{17})\). In the second case, besides \( f_1 \) and \( f_2 \) becoming common files, \( f_3 \) also became a common file, and it was also included in the hierarchy. Figure 34 shows the common files section of the TO-ISA output for both experiments.

```
# II. COMMON_FILES SECTION  (using \( \eta = 0.7 \))

# A. INDEPENDENT FILES

# B. FILE HIERARCHIES

FH-1 : S1.1(3) S2.1(3)
  f1
  f2  S1.1(3) S2.1(3) S3.1(6) S3.2(4) S4.1(2)
  f17 S1.1(3) S2.1(3) S5.1(2)

# II. COMMON_FILES SECTION  (using \( \eta = 0.9 \))

# A. INDEPENDENT FILES

# B. FILE HIERARCHIES

FH-1 :
  f1
  f2  S1.1(3) S2.1(3) S3.1(6) S3.2(4) S4.1(2)
  f3  S3.1(6) S5.1(2)
  f17 S1.1(3) S2.1(3) S5.1(2)
```

Figure 34. Common file sections from experiments with \( \eta \)

The third set of experiments targeted the parameter \( \mu \) (default value \( \mu = 0.8 \); range \( 0.5 \leq \mu < 1 \)). This parameter controls the formation of the hierarchy of file implications. It defines
the range of accepted values for the confidence of a file implication. The default value of $\mu$ means that the file implications in the hierarchy need to have a confidence of at least 80%. Intuitively, as $\mu$ increases the hierarchies will have fewer elements. We ran three experiments using the following parameter values: (1) $\mu = 0.6$ and $\eta = 0.7$, (2) $\mu = 0.9$ and $\eta = 0.7$, and (3) $\mu = 0.9$ and $\eta = 0.9$. We did not use the default value of $\eta$ because it does not generate a hierarchy of file implications. The first two experiments produced identical results. They produce the hierarchy of file implications described in the top part of Figure 34. The third experiment produced an output identical to the bottom part of Figure 34. The interpretation of these results is that the implications $f_2 \rightarrow f_1$, $f_3 \rightarrow f_1$, and $f_{27} \rightarrow f_1$ have confidences larger that 90%. Indeed, these confidences are 100% as shown in Table 6. In other words, every program in the alpha set that uses file $f_2$ also uses file $f_1$. A similar statement is valid for files $f_3$ and $f_{27}$.

The last set of experiments focused on the parameter $\lambda$ (default value $\lambda = 0.3$; range $0 < \lambda \leq 0.5$). The parameter $\lambda$ controls the final hierarchical arrangement of main subsystems. It sets the range of accepted values of file coverage that a main subsystem has to have to be the parent of another main subsystem. The expected behavior is that as $\lambda$ becomes larger more main subsystems form part of the final hierarchy. We ran two experiments using $\lambda = 0.4$ and $\lambda = 0.5$. Both experiments produced the same hierarchy of main subsystems, which is shown in Figure 35. The figure shows that the main subsystems were arranged in one hierarchy. $S_5$ is a subsystem of $S_3$, $S_3$ is a subsystem of $S_4$, and $S_4$ a subsystem of $S_2$. This final arrangement of main subsystems left subsystem $S_1$ out of the hierarchy. This result derives from the logic of the algorithm that builds the hierarchies. It establishes hierarchical relationships among subsystems that have programs that use a similar set of files. Thus, $S_1$ was not related to any other subsystem because it contains many files that are used only inside $S_1$ (see Table 6).
The results of these experiments show that ISA produces similar subsystem decompositions if the confidences of the associations are high. These types of associations are more likely to be present in systems that have a monolithic design such as the TRS. If we consider the results of the default subsystem decomposition, S1 and S3 seem to be the most important subsystems in the TRS system. S1 contains more files than any other subsystem and all the programs in S1 use most of these files. Similarly, S3 contains more programs than any other subsystem. In addition, S3 can stand alone as an independent system because it uses no external files. The rest of the subsystems seem to play a secondary role because the programs in these subsystems use primarily four files (f1, f2, f3, and f27) or files that are used few times (most likely temporary files). In addition, all the experiments highlighted the relevant role that f1, f2, f3, and f27 play in TRS. This fact is easily visualized in the RM diagrams.

5.3. Case Study: The PS6 System

The second case study is a 12-year-old point of sale system (PS6). PS6, written in COBOL, runs on a Unix environment on a personal computer. PS6 consists of approximately 220,000 lines of code divided into 89 source code files. Each one of these source code files is considered an independent program. In addition, PS6 has approximately 400 user-defined "copy" files. These files resemble the "include" files from the C language. As with the TRS system, the data repositories defined by a "SELECT" statement are considered files.
The approach to this case study was similar to the approach used for the TRS system. First, we applied the ISA methodology to decompose the PS6 system into subsystems and to produce the TO-ISA report. We used the default values of all the parameters for this initial analysis. Next, we ran several experiments. In each experiment, we vary the value of one or two of the parameters. The outcome of each experiment was compared with the default subsystem decomposition to determine the influence of each parameter in the resulting subsystem decomposition. The default subsystem decomposition is the decomposition obtained by using the default value of all the parameters. We used the RE-ISA tool to run all the experiments.

The first run corresponds to the default subsystem decomposition, which uses the following default parameter values: $\beta = 1$, $\gamma = 1$, $\sigma = 3$, $\delta = 0.5$, $\tau = 0.3$, $\varepsilon = 0.4$, $\eta = 0.5$, $\mu = 0.8$, and $\lambda = 0.3$. We used $\sigma = 3$ instead of $\sigma = 2$ to reduce the number of redundant associations. The subsystem decomposition process, including the parsing of the source code, took 65 seconds on a Digital Alpha Workstation.

The resulting TO-ISA representation is given in the Appendix. The report shows that the PS6 system contains 89 programs and 82 files. Also, 25 programs (28%) use less than two files. These programs were not included in the alpha set, thereby they are listed under the unconnected components section (i.e., section V of TO-ISA). Similarly, 43 files (52%) are used by only one program. Thus, they were not included in the alpha set and are regarded as unconnected files. In summary, the alpha set contains 64 programs and 39 files. The resulting subsystem decomposition contains 64 programs and 39 files organized in several subsystems.

The PS6 system does not appear to have a monolithic design because it contains several programs that use many files and several files that are used by many programs. Program p62 is the program that uses more files. It uses 24 files. Moreover, 16 programs use 10 files or more. Table A in Figure 36 lists these programs and table B provides the number of programs that
use less than 10 files grouped by number of files that the programs use. These tables show that PS6 contains a balance of programs in terms of the number of files they use. There are 16 programs that use 10 or more files. Many of these programs use between 15 and 19 files. These programs represent 25% of the programs in the alpha set as shown in table B in Figure 36. Moreover, most of the programs (55%) use between 4 and 9 files and just 20% use 3 or 2 files.

<table>
<thead>
<tr>
<th>No</th>
<th>PgmID</th>
<th>Files</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>62</td>
<td>24</td>
</tr>
<tr>
<td>2</td>
<td>79</td>
<td>21</td>
</tr>
<tr>
<td>3</td>
<td>78</td>
<td>19</td>
</tr>
<tr>
<td>4</td>
<td>76</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>17</td>
</tr>
<tr>
<td>6</td>
<td>32</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>36</td>
<td>17</td>
</tr>
<tr>
<td>8</td>
<td>2</td>
<td>16</td>
</tr>
<tr>
<td>9</td>
<td>72</td>
<td>16</td>
</tr>
<tr>
<td>10</td>
<td>73</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>12</td>
<td>59</td>
<td>11</td>
</tr>
<tr>
<td>13</td>
<td>1</td>
<td>10</td>
</tr>
<tr>
<td>14</td>
<td>18</td>
<td>10</td>
</tr>
<tr>
<td>15</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>16</td>
<td>55</td>
<td>10</td>
</tr>
</tbody>
</table>

Table A

<table>
<thead>
<tr>
<th>No</th>
<th>FileID</th>
<th>Pgmns</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>54</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>33</td>
</tr>
<tr>
<td>3</td>
<td>37</td>
<td>28</td>
</tr>
<tr>
<td>4</td>
<td>42</td>
<td>28</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>27</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>7</td>
<td>5</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>41</td>
<td>22</td>
</tr>
<tr>
<td>9</td>
<td>8</td>
<td>21</td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>11</td>
<td>44</td>
<td>14</td>
</tr>
<tr>
<td>12</td>
<td>9</td>
<td>13</td>
</tr>
<tr>
<td>13</td>
<td>15</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>43</td>
<td>13</td>
</tr>
<tr>
<td>15</td>
<td>38</td>
<td>12</td>
</tr>
<tr>
<td>16</td>
<td>40</td>
<td>11</td>
</tr>
<tr>
<td>17</td>
<td>17</td>
<td>10</td>
</tr>
</tbody>
</table>

Table C

<table>
<thead>
<tr>
<th>Number of Programs</th>
<th>Number of files used</th>
<th>%</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>10 or more</td>
<td>25%</td>
</tr>
<tr>
<td>4</td>
<td>9</td>
<td>6%</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>3%</td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>14%</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>16%</td>
</tr>
<tr>
<td>10</td>
<td>4</td>
<td>16%</td>
</tr>
<tr>
<td>8</td>
<td>3</td>
<td>13%</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>7%</td>
</tr>
</tbody>
</table>

Table B

Figure 36. Tables of program and file usage in the PS6 system

With respect to the files, the PS6 system contains several files that are used by most of the programs. File f3 is the file that is used the most. A total of 54 programs (84%) use it. File f4 is the next most used file with 33 programs (53%) accessing it. Furthermore, 17 files are used by 10 or more programs. These 17 files represent the 43% of the files in the alpha set. This large percentage suggests that the PS6 system has several central repositories. Table C in Figure 36 lists these files along with the number of programs that use them. The table can be divided in two clusters of files. The upper 10 files are used by 18 programs or more while the
lower part contains files that are used by 14 programs or less. These highly used files are expected to be classified as common files by the ISA methodology. The remaining 22 files can be divided into three groups according to the number of programs that access them. There are 7 files that are used by 7 to 9 programs, 7 files that are used by 4 to 6 programs, and 8 files that are used by 3 or less programs. These low-used files may support particular functions or may serve as communication buffers among programs. In summary, the PS6 system has a diverse and rich organization that suggest multiple interrelationships among its programs and files as opposed to the TRS organization where most of the programs access few files. Thus, the ISA methodology may decompose the PS6 system in different ways according to the values of the parameters. This characteristic is very desirable to test the effect of each parameter in the final subsystem decomposition.

The default decomposition of the PS6 subsystem produced the following results (see the Appendix). From the 64 programs in the alpha set, 56 (87%) were assigned to subsystems and the 8 remaining programs were classified as singular programs. From the 39 files in the alpha set, 31 files (80%) were assigned to the subsystems and the 8 remaining files (20%) were classified as common files. From the 8 common files, 7 files formed a hierarchy of file implications and one file was classified as independent file. Intuitively, f3 is the best candidate to be the root of the hierarchy of file implications because f3 is the most used file. Indeed, the TO-ISA report shows that the root of the file hierarchy is file f3. Files f7, f6, f37, f42, and f43 are the children of the root. In addition, f8 is the child of f7. Thus, the deepest level of the file hierarchy is two. One example of the meaning of this hierarchy is that at least 80% of the programs (\(\mu = 0.8\)) that use file f8 also use file f7 and f3. In general, at least 80% of the programs that use any of the files f7, f6, f37, f42, and f43 also use f3. This information has many potential uses in maintenance activities.
In regard to the subsystem decomposition, the outcome of the analysis is as follows. ISA decomposed the PS6 system into 17 primitive subsystems and 9 complex subsystems. The subsystems were arranged into 7 main subsystems. The number of programs and files assigned to each main subsystem is given in Table 7. As the table shows, main subsystems S3, S4, and S6 are the larger main subsystems. They concentrate 65% of the programs and 70% of the files in the alpha set. This decomposition may suggest that the PS6 system performs three major functions. The other 4 subsystems are small, thereby they might implement auxiliary functions. This observation is supported by the fact that only one file is assigned to these subsystems. Thus, these subsystems access mainly common files or files assigned to other subsystems. Table 7 also indicates that two main subsystems contain sub-subsystems (i.e., S4 and S6 have a hierarchy with more than one level). Although main subsystem S3 contains many files and programs, they are distributed in only two primitive subsystems (i.e., S3.1. and S3.2). Therefore, S3 has only one hierarchical level. Figure 37 shows graphically the hierarchical decomposition of PS6.

Table 7. Programs, files, and subsystems for each main subsystem in PS6

<table>
<thead>
<tr>
<th>Main subsystems</th>
<th>Number of programs</th>
<th>Number of files</th>
<th>Number of primitive subsystems</th>
<th>Number of complex subsystems</th>
<th>Number of levels in hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S3</td>
<td>12</td>
<td>14</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S4</td>
<td>12</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>S5</td>
<td>4</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>S6</td>
<td>18</td>
<td>13</td>
<td>6</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>S7</td>
<td>5</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

The final part of the TO-ISA report is a hierarchical arrangement of the main subsystems. It shows that the main subsystems were organized in two hierarchies. One hierarchy has S6 in the root and S1 in the next level as its only child. The other hierarchy has two levels. S3 is in the root with S7 as its only child in level 1. The second level of the hierarchy contains S2, S4,
and S5 as the children of S7. This hierarchy has four levels because S4 is in the deepest level and it has two levels of its own. Consequently, if the two hierarchies of main subsystems became the children of a single “system root,” the final hierarchical arrangement of the PS6 system will have 5 hierarchical levels. Figure 38 portrays this information graphically.

Figure 37. Hierarchical decomposition of the PS6 system

The next part of the case study is the experimentation with different parameter values to investigate the influence of the parameters in the final subsystem decomposition of the PS6 system. The procedure for this part of the case study is different from its counterpart in the TRS analysis. For the PS6 system, we decided to focus on some metrics instead of observing all the changes produced by varying the value of one parameter. The reason for this decision is that the PS6 system is considerably more complex than the TRS system, therefore a change in a parameter may produce a totally different subsystem decomposition. In other words, comparing subsystem decompositions may involve comparing several TO-ISA reports, which is very complex. Hence, it is easier to compare metrics instead of a complete report.
Figure 38. Hierarchical organization of main subsystems in PS6

We selected the following 14 metrics:

(1) Number of independent files.
(2) Number of files in hierarchies of file implications.
(3) Number of common files.
(4) Number of files in subsystems.
(5) Number of hierarchies of file implications.
(6) Deepest level of the hierarchies of file implications.
(7) Number of main subsystems.
(8) Number of primitive subsystems.

(9) Deepest hierarchical level among main subsystems

(10) Number of main subsystems whose hierarchy contains the deepest level (i.e., metric 9)

(11) Number of programs in subsystems

(12) Number of singular programs

(13) Number of hierarchies of main subsystems (i.e., section vi of TO-ISA)

(14) Deepest level among hierarchies of main subsystems

We selected these metrics because they summarize the information in the major parts of
the TO-ISA report. In addition, they are easy to compute and compare. Although they do not
provide details regarding the differences among the results of the experiments, they can be
used to detect major changes, patterns, and unexpected results.

We ran 17 experiments. For each experiment we produced the corresponding TO-ISA
report and collected the 14 metrics. The first experiment corresponds to the default subsystem
decomposition of the PS6 system (i.e., using the default value of the parameters). The next
three runs focused on δ. We used values of δ = 0.7, δ = 0.85, and δ = 0.90 while the rest of the
parameters maintained their default value. A similar process was used for the rest of the
experiments. Table 8 summarizes the settings and results of the experiments. Column one
contains the experiment number, column two contains the value of the parameters, and the rest
of the columns contain the value of the metrics. In column two, only the parameters that took a
value different from theirs default value are listed.

Experiments 2 to 4 targeted the parameter δ (default value: δ = 0.5). This parameter
controls the associations that enter the analysis. The larger this parameter the fewer
associations considered. Thus, it is expected that as δ increases, the number of programs
assigned to subsystems decreases. As Table 8 shows, this tendency is observed in the PS6
system. The number of programs in subsystems decreased from 56 (using δ = 0.5) to 16 (with
\[ \delta = 0.90 \]. Consequently, the number of singular programs increased, the hierarchy of subsystems flattened (i.e., one-level hierarchies), and the number of main and primitive subsystems decreased.

Table 8. Metrics from the experiments on PS6

<table>
<thead>
<tr>
<th>Run No.</th>
<th>Values of the parameters</th>
<th>No. of independent files</th>
<th>No. of files in FHs</th>
<th>No. of common files</th>
<th>No. of files in subsystems</th>
<th>No. of FHs</th>
<th>Number of main subsystems</th>
<th>Deepest level in hierarchy</th>
<th>No. of main subsystems having deepest level</th>
<th>No. of programs in main subsystems</th>
<th>No. of hierarchies of main subsystems</th>
<th>Deepest level in hierarchy of main subsystems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Default</td>
<td>1 7 8 31 1</td>
<td>2 7 17 2 2</td>
<td>56 8 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>( \delta = 0.70 )</td>
<td>2 9 11 28 1</td>
<td>2 7 16 2 2</td>
<td>51 13 1 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( \delta = 0.85 )</td>
<td>3 11 14 25 1</td>
<td>3 6 9 1</td>
<td>35 29 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>( \delta = 0.90 )</td>
<td>6 18 24 15 4</td>
<td>2 4 5 1</td>
<td>16 48 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( \tau = 0.10 )</td>
<td>2 13 15 24 1</td>
<td>2 10 19 2 1</td>
<td>55 9 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>( \tau = 0.50 )</td>
<td>1 7 8 31 1</td>
<td>2 7 17 2 2</td>
<td>56 8 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>( \varepsilon = 0.50 )</td>
<td>1 7 8 31 1</td>
<td>2 7 17 2 2</td>
<td>56 8 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>( \varepsilon = 0.10 )</td>
<td>1 7 8 31 1</td>
<td>2 7 19 2 2</td>
<td>55 9 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>( \tau = 0.1, \varepsilon = 0.1 )</td>
<td>2 13 15 24 1</td>
<td>2 10 19 2 1</td>
<td>55 9 2 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>( \eta = 0.70 )</td>
<td>3 15 18 21 1</td>
<td>2</td>
<td>same as in default (run 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>( \eta = 0.90 )</td>
<td>3 20 23 16 1</td>
<td>2</td>
<td>same as in default (run 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>( \eta = 0.98 )</td>
<td>3 20 23 16 1</td>
<td>2</td>
<td>same as in default (run 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>( \mu = 0.60 )</td>
<td>1 7 8 31 1</td>
<td>2</td>
<td>same as in default (run 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>( \mu = 0.90 )</td>
<td>4 4 8 31 1</td>
<td>2</td>
<td>same as in default (run 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>( \lambda = 0.50 )</td>
<td></td>
<td></td>
<td>same as in default (run 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>( \lambda = 0.10 )</td>
<td></td>
<td></td>
<td>same as in default (run 1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>17</td>
<td>( \lambda = 0.1, \tau = 0.1 )</td>
<td>2 13 15 24 1</td>
<td>2 10 19 2 1</td>
<td>55 9 1 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This decrement of the number of subsystems and programs in the subsystems caused that fewer files were assigned to subsystems. Therefore, the number of common files increased. In the fourth experiment, four hierarchies of file implications were produced. This result is interesting because the fourth run is the only experiment that produced more than one hierarchy of file implications. Another effect of the parameter \( \delta \) is that as it becomes larger the hierarchical arrangement of main subsystems flattens. In summary, large values of the \( \delta \) parameter reduce the number of programs that are assigned to the subsystems if the
confidences of the associations are not very high. This scenario may produce a poor subsystem decomposition in terms of the number of subsystems produced and the number of levels in the hierarchy of subsystems.

Experiments 5 to 9 evaluated the effect of the $\tau$ and $\varepsilon$ parameters. They control the growth of the subsystems. These parameters are used to evaluate whether a program enters a subsystem. When low values are assigned to these parameters, it is expected that fewer programs would be assigned to the subsystems. Thus, more subsystems would be created or the number of singular programs would increase. Table 8 shows that with the largest possible values for $\tau$ and $\varepsilon$ (runs 6 and 7), a subsystem decomposition identical to the default subsystem decomposition is obtained. However, when $\tau$ or $\varepsilon$ adopt small values (runs 5 and 8) we observe both an increment of the number of primitive subsystems and an increment of singular programs. Interestingly, the number of primitive subsystems increases but the number of programs in subsystems does not increase. This result means that the subsystems contain fewer programs.

Although the parameters $\tau$ and $\varepsilon$ produce similar results, there are some important differences between them. The low value of $\varepsilon$ (run 8) produced fewer main subsystems (i.e., 7) than the main subsystems produced using the same value for $\tau$ (i.e., 10 main subsystems in run 5). Fewer main subsystems caused that more files were assigned to the subsystems. Thereby, run 8 produced fewer common files. Another difference between run 5 and run 8 is that the latter experiment (i.e., $\varepsilon = 0.1$) produced metrics similar to the default run (run 1). The only significant difference is on the number of primitive subsystems. This result suggests that both parameters influence the number of primitive subsystems produced, but $\tau$ seems to influence also the creation of main subsystems. This result is consistent with the design of the ISA methodology because $\tau$ is also used to control the merging of subsystems to create larger
subsystems (e.g., main subsystems). Therefore, the lower the value of \( \tau \), the larger the number of main subsystems. Finally, we ran an experiment to test the influence of both parameters when they adopt low values at the same time (run 9). The outcome is practically the same as in run 5. This result confirms that \( \tau \) has more influence than \( \varepsilon \) on the outcome.

Experiments 10, 11, and 12 tested the effect of \( \eta \) (default value: \( \eta = 0.5 \)). This parameter controls the assignment of files to subsystems. The larger the value of \( \eta \), the more restrictive the file assignment becomes; thereby fewer files would go to the subsystems and more files would become common files. In addition, \( \eta \) has no influence on subsystem formation. Both behaviors are supported by the results of the experiments. Table 8 shows that as \( \eta \) increases, the number of common files also increases, and the metrics about subsystems remain the same. Interestingly, the new common files become part of the hierarchy of file implications. That is, no new common file was classified as independent file. This behavior is not expected in all systems because it depends on the particular usage of each file. Another interesting result is that there is no difference between runs 11 (\( \eta = 0.9 \)) and 12 (\( \eta = 0.98 \)). This means that the files that are assigned to a particular subsystem are mostly used by programs inside it. This result suggests a strong data cohesion within the subsystems.

The next two experiments investigated the effect of the parameter \( \mu \), which has a default value of 0.5. This parameter controls which files can form a parent-child relationship in a hierarchy of file implications. The expected result is that as \( \mu \) becomes larger, fewer files will enter the hierarchy of file implications. Thus, more files will be classified as independent files. In addition, \( \mu \) should have no effect on subsystem formation. Both expected behaviors were confirmed by the results of the experiments 13 and 14 shown in Table 8. Indeed, as \( \mu \) becomes larger, fewer files entered the hierarchy of file implications. In this particular case, varying \( \mu \)
from 0.5 to 0.6 made no difference. However, a more restrictive 0.9 produced the expected results.

The last two experiments targeted the parameter $\lambda$, which controls the hierarchical organization of main subsystems. The expected result is that as $\lambda$ decreases, the fewer levels in the hierarchy of main subsystems are produced. Thus, if $\lambda$ takes a high value (i.e., the restriction weakens) then the hierarchy of main subsystems will contain more levels. This expected behavior is confirmed by the results of experiments 15 and 16 showed in the Table 8. Finally, we ran another experiment that involved a larger number of main subsystems with the expectation that more hierarchical levels would be produced. We ran experiment 17 using a high value for $\lambda$ to allow the formation of several levels, and a low value for $\tau$ to guarantee the generation of many main subsystems. The results of the experiment confirmed the expected behavior. The PS6 system was decomposed into 10 main subsystems, which were organized in a single 3-level hierarchy.

In summary, the outcome of the experiments confirmed the expected behavior of the ISA methodology. The effect of each parameter in the final subsystem decomposition was consistent with its theoretical expected behavior. Although the parameters control some aspects of the subsystem decomposition, it is clear that each particular system will produce a particular family of subsystem decompositions because the final arrangement of programs and files depends largely on the particular architecture of the system (i.e., the specific relationships among the programs and files that form the system).

5.4. Discussion

The approach we adopted to validate the hypothesis of this research has three steps.

(1) Develop a general method to apply data mining to reverse engineering and maintenance problems
(2) Use the method to create a specific methodology that uses data mining to produce a high-level abstraction that can be used in the reverse engineering and maintenance domains.

(3) Validate the methodology with real-world systems

The logic to validate the hypothesis works backwards. If we show that the methodology works with real-world systems, it would mean that the methodology does produce a high-level abstraction that can be used in reverse engineering activities. Consequently, the general method on which the methodology is based on would be validated. Finally, if the method is validated, it would suggest that the hypothesis is correct. Therefore, if we validate the ISA methodology against real systems, we would provide positive evidence that the hypothesis of this research is true.

The results of the two case studies presented in this chapter show that the ISA methodology does produce the outcome that ISA was expected to produce. In addition, the information in this chapter shows that ISA is a fully automatic methodology, that is capable of analyzing large real-world systems, and that produces a high-level abstraction of the subject system. In other words, this chapter provides the information that validates the ISA methodology and that demonstrates that ISA satisfy its design objectives.

Consequently, the results presented in this chapter provide positive evidence that the hypothesis of this research is true. Furthermore, the information in this chapter provides evidence not only that data mining can be used in reverse engineering, but also that that data mining techniques can be a valuable tool for reverse engineering and maintenance. Specifically, the results of this research confirm that data mining techniques are useful for design recovery.

The case studies produced other interesting results. The running times of the analysis of the TRS and the PS6 systems provide pragmatic evidence that ISA is scalable. RE-ISA took
12 seconds to analyze the TRS system, which contains approximately 25,000 lines of code. In the case of the PS6 system, RE-ISA took 65 seconds to complete the analysis of the 220,000 lines of code in PS6. Thus, while the problem size grew 9 fold (i.e., from 25 KLOC to 220 KLOC), the running time only grew 5 fold (i.e., from 12 seconds to 65 seconds). These pragmatic results also showed that the bottleneck of the ISA methodology is the data mining phase. Indeed, most of the RE-ISA running time was devoted to mining the associations. One of the advantages of ISA is its modular design, therefore any fastest data mining algorithm can be plugged into RE-ISA and the performance will improve.

The results of the experiments in both the TRS and the PS6 system suggest that two major factors influence the final subsystem decomposition: the confidence of the mined associations, and the values chosen for the 9 ISA parameters. The confidence of the associations depends on the relationships among programs and files; thereby, the confidence of the associations depends on the architecture of the subject system. Hence, this factor constitutes the “personality” of the system and cannot be manipulated by the analyst. However, the analyst can manipulate the second factor: the selection of the value of the parameters.

We believe that a unique “correct” subsystem decomposition of a system does not exist. We think that the correctness of the decomposition depends on the purpose of it. A decomposition that is perfect for generation of test cases may not be the perfect decomposition for object identification. Therefore, the analyst can use the value of the parameters to fine-tune the subsystem decomposition to produce the “perfect” decomposition that suits the specific objectives of the analysis (e.g., design recovery, generation of test suits, configuration management, object recovery, documentation). In this direction, it is necessary to define specific metrics to determine the “correctness” of a subsystem decomposition given the particular domain in which the decomposition is used. Finally, the results of the experiments
presented in this chapter suggest that the resulting subsystem decomposition produced by ISA is influenced more by the architecture of the system than by the value of the parameters. That is, ISA does not override the architectural structure of the system. Indeed, the influence that the parameters have on the subsystem decomposition is limited; the intrinsic architectural structure of the system is preserved.
CHAPTER 6. SUMMARY AND CONCLUSIONS

The research objective of this work is to show that data mining can be used in reverse engineering and maintenance. We defined a methodology that decomposes a software system into a hierarchy of data cohesive subsystems. In other words, this methodology, called ISA (Identification of Subsystems based on Associations), provides evidence that this research reached its objective because ISA produces a reverse engineering artifact based on data mining techniques. The results of this research show not only that data mining can be used in the reverse engineering domain, but also that reverse engineering can benefit from using data mining technology.

This research has been divided in four major sections corresponding to Chapters 2 to 5 in this document: background, ISA methodology, graphical representation model (RM), and experiments. Chapter 2 presented the background information for this research. It included a discussion of the main concepts in the data mining and design recovery domains and a review of literature of related research works. In addition, Chapter 2 described the motivation for pursuing a data mining approach to design recovery. Moreover, it included the definition of the three-step (TS) method, which is a general framework to apply data mining in the reverse engineering domain.

Chapter 3 described the ISA methodology. It included the motivations, requirements, and development of the ISA design. In addition, chapter 3 gave a detailed description of ISA, which included a description of the input and output of ISA, a full description of each of the 11 steps of ISA, a pseudocode specification of all the algorithms, and a detailed description of the textual representation of the ISA outcome (i.e., the TO-ISA report).

Chapter 4 was dedicated to the definition of the graphical representation of the ISA output, called RM (Representation Model). In this chapter, we defined all the graphic elements required to visually represent the output of ISA. Moreover, the chapter included a description
of the motivations and design objectives for RM as well as a review of literature of related research.

Finally, chapter 5 included a description of the use of ISA to analyze actual software systems. This chapter included the description of the RE-ISA tool, which implements the ISA methodology, as well as two case studies. For each case study, we included a description of the subject system, a description of the results obtained with ISA, and a series of experiments designed to understand the influence of each of the ISA’s parameters in the resulting subsystem decomposition.

Sections 6.1 to 6.3 address the conclusions, the contributions of this research, and the different research directions for future work, respectively.

6.1. Conclusions

The main conclusion of this research is the confirmation of the research hypothesis. That is, data mining is a feasible and valuable approach to software reverse engineering and maintenance. In particular, this research shows that data mining can be used for design recovery, as confirmed by the ISA methodology that produces a data cohesive subsystem decomposition of a software system. In addition, this research shows that data mining has the following features that are beneficial to the reverse engineering domain.

(1) Data mining can process large volumes of information. Therefore, data mining techniques are capable of dealing with the complexity of large software systems. This fact is confirmed by the ISA methodology, which is capable of analyzing large systems in acceptable computational time.

(2) Data mining elicits meaningful information without previous knowledge of the domain. This feature is especially important when dealing with software systems that lack documentation. As the ISA methodology shows, data mining techniques can produce relevant information of the subject system without any previous knowledge.
of the system's domain. Indeed, the source code is the only source of information that ISA requires to produce a sound subsystem decomposition.

(3) Data mining extracts novel non-trivial relationships. Reverse engineering use this capability of data mining to deal with the multiple interrelationships among the components of a software system. In addition, reverse engineering can use the discovered knowledge to improve the information and knowledge extracted from a subject system. This data mining feature is confirmed by the ISA methodology. In this case, besides producing the information to guide the subsystem decomposition, data mining generated hierarchies of file implications, which constitutes a novel type of knowledge about the system. In data mining, the larger the data set, the better the possibility that the mining process produces meaningful new information. Thus, the larger the subject system, the larger the possibility of uncover relevant relationships among system components.

(4) Data mining is automatable. Reverse engineering techniques have to be automated to be of any practical use. Non-automatic techniques are useless to analyze large real-world systems. Therefore, automation is an important characteristic of data mining for its use in the reverse engineering domain. The ISA methodology and its implementation (i.e., the RE-ISA tool) show that data mining can be the foundation of a methodology that can be completely automatic.

These conclusions validate our motivations to pursue a data mining approach to reverse engineering. In addition, they support the conclusion that the data mining has a potential to be used in other areas of software engineering. However, the data mining approach to reverse engineering has important limitations. First, to be able to use data mining it is necessary to have an initial data set. The definition of this data set, which we call a database view of the system, is not straightforward. Some craftsmanship is required to identify the system
components that go into the data set and to define the appropriate representation of the data set. In addition, it is necessary to identify the right combination of the data mining algorithms that can be applied to the data set and the software artifact that can be produced with the mined information. We conclude from this research experience that the critical point to using data mining in reverse engineering is the definition of the database representation. This conclusion imposes a fundamental restriction to the approach presented in this document: to profit from the TS model, there must be a way to decompose the subject system into components that can be represented in a database.

Other conclusions from this research concern the ISA methodology. First, ISA meets its design objectives and produces what it was designed to produce. In other words, ISA produces a hierarchical data-cohesive subsystem decomposition of a software system. Moreover, practical results confirm that the behavior of ISA is consistent with its expected theoretical behavior when the values of the ISA parameters vary. In addition, we conclude that ISA does not distort the architectural structure of the system. The influence that the parameters have on the subsystem decomposition is limited; the intrinsic architectural structure of the system is preserved. In other words, the resulting subsystem decomposition produced by ISA is influenced more by the architecture of the system than by the value of the parameters. Nevertheless, the analyst can manipulate the parameters to produce the most convenient subsystem decomposition that suits his or her specific reverse engineering or maintenance activity. The analyst can try different parameter values to produce different effects on the subsystem decomposition such as stressing a highly data cohesive subsystem decomposition, generating many subsystems, decreasing the number of singular programs, and avoiding or stressing flat hierarchies. We believe that a unique "correct" subsystem decomposition of a system does not exist. We think that the correctness of the decomposition depends on the purpose of it. A decomposition that is perfect for generation of test cases may not be the
perfect decomposition for object identification. In this direction, it is necessary to define specific metrics to determine the "correctness" of a subsystem decomposition given the particular domain in which the decomposition is used.

ISA can be applied to any system that satisfies the following conditions. It should be possible to decompose the system into relatively independent units of source code (i.e., programs), the system should contain several data repositories (i.e., data files), and the units of code should access information on the data repositories. ISA produces better results when the system contains a large number of programs, a large number of files, and when most of the programs access many files. This description matches many COBOL, RPG, and PL/1 legacy systems. Normally, this type of systems is composed of a set of programs and a set of data repositories. For each program, there is a separate source code file. In addition, the data repositories are implemented as flat files, tables in a relational database, or files in a hierarchical database system.

Significant benefits of the ISA methodology are that it is automatic and scalable. ISA is specified at enough detail to allow its automation. The RE-ISA tool implements the ISA methodology and is capable of producing automatically a subsystem decomposition of a software system. The only required input to RE-ISA is the location of the source code of the subject system. In addition, ISA is scalable in the sense that it can analyze large systems within reasonable computational time. Pragmatic results show that while the problem size grows 9 fold, the running time only grows 5 fold. These pragmatic results also showed that the bottleneck of the ISA methodology is the data mining phase. Indeed, most of the RE-ISA running time was devoted to mining the associations. This result is an advantage of the ISA methodology because it can be implemented using the best data mining algorithm at hand. Automation and scalability set ISA apart as a methodology capable of analyzing not only theoretical academic examples but also large real-world legacy systems.
Another advantage of the ISA methodology is its consistency. ISA always produces the same results if it is fed with the same parameters. In addition, ISA always produce a subsystem decomposition if there are at least two programs in the system that do not use the same set of files.

The principal limitation of the ISA methodology is that it only can be used in systems that can be viewed as a collection of several relatively independent portions of code (i.e., programs) and several data repositories (i.e., data files). The programs and data files can be defined logically. For example, if the all the source code of the system is in a single file, it may be possible to identify several modules within this file. In that case, those modules would become the “programs.” However, if the system cannot be decomposed into several “programs” and “files,” then ISA cannot be used. In addition, ISA may not produce significant results in monolithic-type systems in which most of the programs access the same set of files. Moreover, ISA may not produce a subsystem decomposition if the system contains many programs that use few files (i.e., 1 to 3 files), or if the systems contains many files that are used by only one program. Finally, ISA may not create a hierarchy of subsystems if the system contains many programs and few files, or few programs and many files.

We also detected that to facilitate the manual drawing of RM diagrams the TO-ISA report should be modified to include the number of external programs that use an internal file.

The results of the ISA methodology can be used in many domains. The subsystem decomposition produced by ISA can be used for configuration management. For example, the recovered structure can be used as the “official” system architecture; thereby, any maintenance activity on the system should not modify this architecture. In other words, the “official” system architecture will assure that future maintenance works do not create undesirable interrelationships. In another direction, ISA can be used to obtain the subsystem
decomposition of a system periodically. The decomposition can be used to track the growth of
the system, to compare versions of the systems, or to detect architectural changes.

The ISA methodology and the RM representation can be used for impact analysis. Impact
analysis focuses on identifying the consequences of making a change in the system. The
information produced by ISA facilitates the identification of all the components that are
related to the particular program, file, or subsystem that is going to be modified. In addition,
RM diagrams facilitate impact analysis by providing a graphic map of the system that shows
the relationships among system components. The impact analyst can locate in the RM diagram
the component that is going to be changed and easily identify the components that may be
influenced by the change.

In the same direction, ISA can also be used to define test suits and to define the testing
strategy. ISA basically forms clusters of programs and files. Thus, defining test suits is
simplified because it is possible to isolate specific clusters of programs and files. In addition,
since ISA provides the relationships of each component in the system, it is easy to identify all
the system components related to a particular cluster of programs and files, thereby facilitating
the customization of the test suit. Moreover, the subsystem decomposition can be controlled to
maximize the internal cohesiveness of the clusters, thereby minimizing the external
relationships and facilitating the testing. The information produced by ISA also can be used to
define the testing strategy of the system or the parts of the system that have been changed. The
tester can easily identify important information such as the critical files in the system, the
main relationships within a subsystem, the main relationships among subsystems, the scope of
a file in a hierarchy of subsystems, and the files that are strongly related to other files (e.g.,
files in a hierarchy of file implications).

In general, ISA is a methodology that conforms to the object-oriented paradigm. One of
the main concepts of the object-oriented paradigm is encapsulation or information hiding. One
aspect of this concept is that data should be encapsulated along with the code that manipulates it. In this sense, ISA produces a subsystem decomposition that maximizes encapsulation because ISA puts in the same subsystem data files and the programs that manipulate them. In other words, ISA promotes data-cohesiveness, which means that the programs are grouped based on the files they use. The larger the data-cohesiveness within a subsystem the better the subsystem meets the encapsulation principle. For example, if all the files assigned to subsystem X are used only by programs in subsystem X, then X meets the encapsulation principle because it contains the data and the functionality that manipulates it.

Finally, we summarize our conclusions regarding the graphical representation model RM. RM offers a high-level visual representation of the subject system that facilitates its understanding. Moreover, RM diagrams are capable of representing all the information produced by the ISA methodology. RM offers an easy-to-follow representation of the identified subsystems and their components. RM diagrams facilitate the detection of highly used files, subsystem interrelationships, and file relationships. In addition, RM makes easier the identification of the critical components of the system as well as the analysis of the system at several levels of abstraction. Therefore, RM can be used at several levels of a reverse engineering analysis or a maintenance effort. Another advantage of RM is its simplicity. It uses few building blocks and simple semantics. Thus, RM diagrams can be drawn with off-the-shelf diagramming tools. Despite its simplicity, RM is capable of representing hierarchies of file implications, which increase the expressiveness of the diagrams. Finally, RM also can be used to analyze and document transactional software systems that use large quantities of flat files or a relational database with many tables.

In summary, this research has demonstrated the value of data mining in the maintenance domain. In addition, two products of this research, the ISA methodology and the RM model,
can be used not only for reverse engineering but also in other maintenance activities such as configuration management, impact analysis, and testing.

6.2. Contributions

The main contribution of this research is that it provides evidence not only that data mining can be used effectively in reverse engineering, but also that data mining is a valuable tool in this domain. The results of this research suggest that data mining can be applied to different problems in reverse engineering besides subsystem decomposition. Hence, this research opens a promising research area: reverse engineering and data mining. This research shows that reverse engineering can profit from certain data mining features. Data mining can elicit meaningful relationships among elements in a data set, can uncover relevant information without any previous knowledge of the data in the data set, and can handle large volumes of information. Most of all, these features can be delivered automatically. These features of data mining can be used to address the reverse engineering problems derived from the complexity, the lack of documentation, the size, and the lack of structure of large software systems. Thus, a major contribution of this research is the identification of a new approach to reverse engineering and the demonstration of the feasibility of the approach.

Theoretical contributions of this research include the interpretation of an association in the context of reverse engineering, the definition of clustering algorithms and distance metrics based on association rules, the general three-step method to apply data mining to reverse engineering and maintenance, the ISA methodology and all the algorithms that implement it, and the definition of the graphical model RM.

One of the key contributions and a essential idea of this research is the interpretation of an association in the context of reverse engineering. An association has the form \( s[x_1, x_2, x_3, \ldots, x_n] \) where \( s \) is the support of the association (i.e., the number of tuples that contain \( x_1, x_2, x_3, \ldots, x_n \)) and \( x_1, x_2, x_3, \ldots, x_n \) are the elements in the association. The mining of associations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
are derived from basket analysis. In that context, the association means that products $x_1$, $x_2$, $x_3$, ...

$\ldots$, $x_n$ are found together in $s$ transactions, where a transaction contains all the products a customer buys in a visit. Thus, the meaning of an association in its original context is that the products that are in an association that has a large value of $s$ tend to be bought at the same time. This information is used to define the floor plan of the stores, to segment the market, or to improve marginal profits. This scenario is very different from a reverse engineering scenario. Nevertheless, the right interpretation of an association rule translated into a direct application of data mining techniques to reverse engineering. If instead of a transaction we use a tuple, if instead of products we use programs, and if instead of clients we use files, then an association $s[x_1, x_2, x_3, \ldots, x_n]$ represents a set of programs that use the same $s$ files. The association does not mean that all the programs in the association use only the $s$ files. For example, $x_1$ may use more than $s$ files and $x_2$ may use exactly $s$ files. Interpreting an association as the set of programs that use the same data repositories was used to guide a clustering process that puts in the same cluster the programs that use the same data repositories.

The interpretation of an association rule described above is the underlying idea of another contribution of this research: the definition of clustering algorithms and distance metrics. We defined a clustering algorithm, called *junta*, that uses this interpretation of an association to produce a hierarchical clustering. Junta can be classified in the family of clustering algorithms based on centroids. These algorithms cluster elements based on a distance function that calculates the "distance" from the element being clustered to certain predefined elements called centroids. The element is assigned to the cluster that contains the closest centroid. Junta uses associations to do both, identify centroids and calculate distances. The associations with largest support and confidence are used to define the centroids. The distance between two programs is given by a combination of the ratios of the files they use in common (e.g., the
support) and the total files they use. These ratios are obtained from the association that involves those two programs. A similar approach is used to calculate the "distance" between two clusters of programs. In summary, another contribution of this research is the definition of clustering algorithms and distance functions or metrics based on associations.

Another contribution of this work is the three-step (TS) method to apply data mining to reverse engineering and maintenance. The relevance of the TS method is that it can be used as a framework to develop other methodologies. ISA is a particular instantiation of the TS method, therefore other methodologies can be created depending on the specific instantiation of each of the steps of the TS method. For example, it may be possible to define a database view of the system that contains procedures and global variables, mine association rules, and produce an object-oriented decomposition of the subject system. These ideas are explored in more detail in section 6.3.

The major technical contribution of this research is the ISA methodology and the algorithms that implement it. ISA is relevant because it produces sound reverse engineering artifacts without any feedback from the user. That is, ISA can recover a design automatically. In addition, ISA is capable of processing large software systems. These features of ISA set it apart from academic design recovery methods and techniques that lack a real potential for industrial use. In addition, ISA is a major contribution because it demonstrates that data mining is a valuable tool for reverse engineering.

Finally, this research produced the graphical model RM. The importance of this model is that it can be used not only to represent the outcome of ISA, but also to represent, analyze, design, and document any system that uses many data repositories shared by many programs.

6.3. Future Work

This research explored a new approach to reverse engineering. However, this research is only a sample of the many ways that data mining can support reverse engineering and
maintenance activities. Consequently, there are many open issues that can be explored in future research. We have identified several research lines in the following two directions:

1. To continue this research to improve its products
2. To apply the ideas of this research to develop new methodologies

Open issues regarding future work that focus on improving this research can be classified in three areas: improvements to the TS method, improvements to the ISA methodology, and improvements to the RM model.

The current definition of the TS method assumes a static analysis of the subject system. An improvement to the TS method would be to incorporate dynamic analysis to this framework. For example, step one of TS calls for the definition of a database view of the subject system. However, the database could be created not as a view of the system but as a log of the behavior of the system. In other words, the database would contain dynamic information about the system. Thus, a direction for future research is the definition of an improved TS method or of a new framework capable of profiting from dynamic analysis. This new model would produce a different family of methodologies.

In regard to the ISA methodology, there are several opportunities for improvement. ISA could be enhanced to form hierarchies of file implications within subsystems. The current version of ISA identifies the common files within a subsystem, but it does not organize those files into hierarchies of file implications. ISA only creates hierarchies of file implications with "global" common files (i.e., files that are not assigned to any subsystem). In another direction, future research work can be focused on the creation of the database view of the subject system. ISA uses the "usage" relationship between programs and files to create the database view. However, this relationship does not take into consideration the type of access to the file. That is, the relationship does not consider if the file is accessed to read, write, delete, or update information on it. Thus, another direction for future research is to use the "type" of access to
the file to create the database view of the system. This approach may require the modification
of some of the steps of ISA and the refinement of the clustering criteria, which may lead to
better subsystem decompositions. Another opportunity for future research is to research how
the concept of evolutionary software can be incorporated into ISA. That is, to define what
happens with the results of ISA when the system changes. Does all the analysis have to be
done again or just part of it? How does ISA have to be modified to incorporate the changes
made to a system without having to run a complete analysis again? These issues are relevant
when working with very large systems that are under constant maintenance.

Finally, the RM model also offers many possibilities for future research. The most
important research area is the refinement of RM to facilitate its automation. This refinement
includes the definition of the process to draw RM diagrams, the addition of drawing elements
and notation, and the formalization of the model. The process to draw RM diagrams must
include specific steps and heuristics to define the layout of the diagrams and to determine
which elements of the ISA outcome are represented in each diagram. For example, the process
should define the procedure to decide whether a file-usage relationship is represented in a
high-level or in a low-level diagram. In summary, the major opportunity for future research
work that focuses on RM is the construction of a tool that takes the information produced by
ISA and automatically draws the corresponding RM diagrams. This tool should include
browsing and zooming capabilities.

The second direction for future research work consists of applying the ideas of this
research to develop new methodologies. In this direction, an interesting possibility for future
research is to develop a new methodology for finer-grain analysis. That is, instead of
decomposing a system into subsystems, decompose a module or a program into cohesive sets
of code and data. For example, it may be possible to use the ideas in this research to develop a
methodology that works at program level. The methodology would use data mining to find
relationships between variables (e.g., global variables, parameters, and global data structures) and segments of code (e.g., procedures, subroutines, and functions). Then it would create a decomposition of the program into objects. A variant of this option for future work is to try to apply or adapt the ISA methodology to finer-grain analysis.

Another opportunity for future research is to use data mining functions other than association rules. Algorithms that mine sequential patterns are good candidates to conduct research in this direction. A sequential pattern consists of a set of events that occurs in sequence during certain amount of time. As with association rules, sequential patterns are defined based on confidences. Sequential patterns might be used to identify procedure calling sequences, variables that are used in sequence, or data that is accessed in sequence. This information can be used in different activities such as detection of redundant code, identification of methods, or the design of test suits. Another variant of this idea for future research is to mine different patterns at the same time in the data mining part of the TS method. Thus, there would be more information (i.e., mined patterns) to create better reverse engineering artifacts.

Finally, the results of this research suggest that data mining can be applied to other domains in addition to design recovery. Therefore, an exciting direction for future research is to explore the use of data mining not only in reverse engineering but also in software engineering in general. For example, we foresee the use of data mining techniques in configuration management, testing, metrics, performance analysis, and redundancy detection.
REFERENCES


156

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


[Han95] J. Han, Y. Fu, “Discovery of Multiple-Level Association Rules from Large Databases,” Proc. 21st Int'l Conf. on Very Large Data Bases (VLDB '95), Sept. 1995, pp. 420-431.


159

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.


APPENDIX. TO-ISA REPORT FOR THE PS6 SYSTEM

# I. HEADER SECTION

(a) SYSTEM NAME: PS6
(b) PROCESSING DATE: 22 Jan 1999 08:23:10 GMT
(d) PARAMETERS

\[
\begin{align*}
\text{alpha} &= 2.0 & \text{beta} &= 2.0 & \text{support} &= 3.0 & \text{delta} &= 0.5 & \text{tau} &= 0.3 \\
\text{epsilon} &= 0.4 & \text{eta} &= 0.5 & \text{mu} &= 0.8 & \text{lambda} &= 0.3 \\
\end{align*}
\]

(e) KEY - PROGRAMS

<table>
<thead>
<tr>
<th>ID</th>
<th>Program Name</th>
<th>#files</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>4TO6PS.PS</td>
<td>(10)</td>
</tr>
<tr>
<td>p3</td>
<td>EXPSFU.PS</td>
<td>(17)</td>
</tr>
<tr>
<td>p5</td>
<td>FIXP.SS</td>
<td>(0)</td>
</tr>
<tr>
<td>p7</td>
<td>PSATHO.PS</td>
<td>(1)</td>
</tr>
<tr>
<td>p9</td>
<td>PSBATO.PS</td>
<td>(6)</td>
</tr>
<tr>
<td>p11</td>
<td>PSCFIP.PS</td>
<td>(6)</td>
</tr>
<tr>
<td>p13</td>
<td>PSCIH.PS</td>
<td>(7)</td>
</tr>
<tr>
<td>p15</td>
<td>PSCUHP.PS</td>
<td>(6)</td>
</tr>
<tr>
<td>p17</td>
<td>PSDEVO.PS</td>
<td>(1)</td>
</tr>
<tr>
<td>p19</td>
<td>PSDOCO.PS</td>
<td>(5)</td>
</tr>
<tr>
<td>p21</td>
<td>PSDLPL.PS</td>
<td>(5)</td>
</tr>
<tr>
<td>p23</td>
<td>PSDLAP.PS</td>
<td>(9)</td>
</tr>
<tr>
<td>p25</td>
<td>PSDLRM.PS</td>
<td>(3)</td>
</tr>
<tr>
<td>p27</td>
<td>PSDSCP.PS</td>
<td>(5)</td>
</tr>
<tr>
<td>p29</td>
<td>PSEDCO.PS</td>
<td>(1)</td>
</tr>
<tr>
<td>p31</td>
<td>PSEODD.PS</td>
<td>(6)</td>
</tr>
<tr>
<td>p33</td>
<td>PSFRMM.PS</td>
<td>(6)</td>
</tr>
<tr>
<td>p35</td>
<td>PSFRMP.PS</td>
<td>(1)</td>
</tr>
<tr>
<td>p37</td>
<td>PSHISN.PS</td>
<td>(0)</td>
</tr>
<tr>
<td>p39</td>
<td>PSHSCP.PS</td>
<td>(3)</td>
</tr>
<tr>
<td>p41</td>
<td>PSHSUPU.PS</td>
<td>(1)</td>
</tr>
<tr>
<td>p43</td>
<td>PSHSSP.PS</td>
<td>(5)</td>
</tr>
<tr>
<td>p45</td>
<td>PSITSUP.PS</td>
<td>(4)</td>
</tr>
<tr>
<td>p47</td>
<td>PSIHPUP.PS</td>
<td>(5)</td>
</tr>
<tr>
<td>p49</td>
<td>PSINHP.PS</td>
<td>(6)</td>
</tr>
<tr>
<td>p51</td>
<td>PSINTG.PS</td>
<td>(2)</td>
</tr>
<tr>
<td>p53</td>
<td>PSITHP.PS</td>
<td>(6)</td>
</tr>
<tr>
<td>p55</td>
<td>PSLAYP.PS</td>
<td>(10)</td>
</tr>
<tr>
<td>p57</td>
<td>PSMAL.SP</td>
<td>(1)</td>
</tr>
<tr>
<td>p59</td>
<td>PSENEW5.PS</td>
<td>(11)</td>
</tr>
<tr>
<td>p61</td>
<td>PSPNU.PS</td>
<td>(4)</td>
</tr>
<tr>
<td>p63</td>
<td>PSREGLP.SS</td>
<td>(5)</td>
</tr>
<tr>
<td>p65</td>
<td>PSREGP.PS</td>
<td>(1)</td>
</tr>
<tr>
<td>p67</td>
<td>PSSHMP.PS</td>
<td>(3)</td>
</tr>
<tr>
<td>p69</td>
<td>PSSTEMP.PS</td>
<td>(1)</td>
</tr>
<tr>
<td>p71</td>
<td>PSSUBN.PS</td>
<td>(0)</td>
</tr>
<tr>
<td>p73</td>
<td>PSTIXB.PS</td>
<td>(18)</td>
</tr>
<tr>
<td>p75</td>
<td>PSTIXD.PS</td>
<td>(0)</td>
</tr>
<tr>
<td>p77</td>
<td>PSTIXG.PS</td>
<td>(4)</td>
</tr>
<tr>
<td>p79</td>
<td>PSTIXM.PS</td>
<td>(24)</td>
</tr>
<tr>
<td>p81</td>
<td>PSTIXO.PS</td>
<td>(6)</td>
</tr>
<tr>
<td>p83</td>
<td>PSTIXS.PS</td>
<td>(2)</td>
</tr>
</tbody>
</table>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
<table>
<thead>
<tr>
<th>ID</th>
<th>File Name</th>
<th># pgms.</th>
<th>ID</th>
<th>File Name</th>
<th># pgms.</th>
</tr>
</thead>
<tbody>
<tr>
<td>f1</td>
<td>COMP-FIL</td>
<td>(5)</td>
<td>f2</td>
<td>MULTI-COMP-FIL</td>
<td>(4)</td>
</tr>
<tr>
<td>f3</td>
<td>PS-STR-FIL</td>
<td>(55)</td>
<td>f4</td>
<td>PS-REG-FIL</td>
<td>(34)</td>
</tr>
<tr>
<td>f5</td>
<td>PS-DRW-FIL</td>
<td>(23)</td>
<td>f6</td>
<td>PS-USR-FIL</td>
<td>(24)</td>
</tr>
<tr>
<td>f7</td>
<td>PS-HDR-FIL</td>
<td>(27)</td>
<td>f8</td>
<td>PS-LIN-FIL</td>
<td>(21)</td>
</tr>
<tr>
<td>f9</td>
<td>PS-HIS-FIL</td>
<td>(14)</td>
<td>f10</td>
<td>INVC-HIST-FIL</td>
<td>(18)</td>
</tr>
<tr>
<td>f11</td>
<td>PATH-FIL</td>
<td>(1)</td>
<td>f12</td>
<td>PS-FRM-FIL</td>
<td>(9)</td>
</tr>
<tr>
<td>f13</td>
<td>PS-EDC-FIL</td>
<td>(10)</td>
<td>f14</td>
<td>PS-EDC-WRK-FIL</td>
<td>(4)</td>
</tr>
<tr>
<td>f15</td>
<td>PS-ACT-FIL</td>
<td>(13)</td>
<td>f16</td>
<td>OE-SHIP-TO-FIL</td>
<td>(7)</td>
</tr>
<tr>
<td>f17</td>
<td>PS-EOD-FIL</td>
<td>(10)</td>
<td>f18</td>
<td>PS-CMP-FIL</td>
<td>(8)</td>
</tr>
<tr>
<td>f19</td>
<td>PS-HUB-FIL</td>
<td>(3)</td>
<td>f20</td>
<td>BTRV-PATH-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f21</td>
<td>BTRV-PS-STR-FIL</td>
<td>(1)</td>
<td>f22</td>
<td>BTRV-PS-REG-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f23</td>
<td>BTRV-PS-DRW-FIL</td>
<td>(1)</td>
<td>f24</td>
<td>BTRV-PS-USR-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f25</td>
<td>BTRV-PS-FRM-FIL</td>
<td>(1)</td>
<td>f26</td>
<td>BTRV-PS-EDC-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f27</td>
<td>BTRV-PS-EDC-WRK-FIL</td>
<td>(1)</td>
<td>f28</td>
<td>BTRV-PS-ACT-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f29</td>
<td>BTRV-PS-HDR-FIL</td>
<td>(1)</td>
<td>f30</td>
<td>BTRV-PS-LIN-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f31</td>
<td>BTRV-PS-HIS-FIL</td>
<td>(1)</td>
<td>f32</td>
<td>BTRV-INVC-HIST-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f33</td>
<td>BTRV-PS-EO-SHIP-TO-FIL</td>
<td>(1)</td>
<td>f34</td>
<td>BTRV-PS-EOD-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f35</td>
<td>BTRV-PS-CMP-FIL</td>
<td>(1)</td>
<td>f36</td>
<td>BTRV-PS-HUB-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f37</td>
<td>IC-CTL-FIL</td>
<td>(28)</td>
<td>f38</td>
<td>WAREHSE-FIL</td>
<td>(12)</td>
</tr>
<tr>
<td>f39</td>
<td>EXTRACT-FIL</td>
<td>(2)</td>
<td>f40</td>
<td>AR-CTL-FIL</td>
<td>(11)</td>
</tr>
<tr>
<td>f41</td>
<td>CUST-FIL</td>
<td>(22)</td>
<td>f42</td>
<td>ITEM-FIL</td>
<td>(28)</td>
</tr>
<tr>
<td>f43</td>
<td>SORT-FIL</td>
<td>(13)</td>
<td>f44</td>
<td>AR-CODS-FIL</td>
<td>(14)</td>
</tr>
<tr>
<td>f45</td>
<td>DEV-FIL</td>
<td>(1)</td>
<td>f46</td>
<td>SLS-MAN-FIL</td>
<td>(8)</td>
</tr>
<tr>
<td>f47</td>
<td>EDC-CFG-FIL</td>
<td>(1)</td>
<td>f48</td>
<td>CASH-LCK-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f49</td>
<td>CASH-TRX-FIL</td>
<td>(3)</td>
<td>f50</td>
<td>INV-LCK-FIL</td>
<td>(2)</td>
</tr>
<tr>
<td>f51</td>
<td>INV-TRX-FIL</td>
<td>(3)</td>
<td>f52</td>
<td>IC-IXT-FIL</td>
<td>(3)</td>
</tr>
<tr>
<td>f53</td>
<td>COMPONENT-COST-FIL</td>
<td>(1)</td>
<td>f54</td>
<td>KIT-FIL</td>
<td>(3)</td>
</tr>
<tr>
<td>f55</td>
<td>LAYER-FIL</td>
<td>(1)</td>
<td>f56</td>
<td>LAYER-WRK-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f57</td>
<td>IC-SER-FIL</td>
<td>(9)</td>
<td>f58</td>
<td>IC-CODS-FIL</td>
<td>(6)</td>
</tr>
<tr>
<td>f59</td>
<td>IC-PRC-FIL</td>
<td>(4)</td>
<td>f60</td>
<td>KEYWORD-FIL</td>
<td>(4)</td>
</tr>
<tr>
<td>f61</td>
<td>PA-CTL-FIL</td>
<td>(2)</td>
<td>f62</td>
<td>PA-SEQ-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f63</td>
<td>STAT-FIL</td>
<td>(7)</td>
<td>f64</td>
<td>VEND-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f65</td>
<td>PS-SITE-LOG-FIL</td>
<td>(1)</td>
<td>f66</td>
<td>PS-MAL-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f67</td>
<td>PASSWORD-FIL</td>
<td>(1)</td>
<td>f68</td>
<td>AR-OPN-ITEM-FIL</td>
<td>(6)</td>
</tr>
<tr>
<td>f69</td>
<td>COMMIS-DUE-FIL</td>
<td>(1)</td>
<td>f70</td>
<td>CASH-APPL-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f71</td>
<td>IC-DIST-FIL</td>
<td>(1)</td>
<td>f72</td>
<td>INV-ACCT-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f73</td>
<td>PA-IVH-FIL</td>
<td>(1)</td>
<td>f74</td>
<td>VAL-GL-ACCT-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f75</td>
<td>AR-ACCT-FIL</td>
<td>(1)</td>
<td>f76</td>
<td>CHNG-CUST-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f77</td>
<td>CL-TAG-FIL</td>
<td>(1)</td>
<td>f78</td>
<td>ALT-ITEMS-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f79</td>
<td>AR-NOTES-FIL</td>
<td>(1)</td>
<td>f80</td>
<td>IC-NOTES-FIL</td>
<td>(1)</td>
</tr>
<tr>
<td>f81</td>
<td>EXP-FIL</td>
<td>(1)</td>
<td>f82</td>
<td>EXP-FIL-2</td>
<td>(1)</td>
</tr>
</tbody>
</table>

END.
# II. COMMONFILES SECTION

# A. INDEPENDENT FILES

f50 : p31(*) p78(*)

# B. FILE HIERARCHIES

<table>
<thead>
<tr>
<th>FH-1</th>
<th>S7.1(1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>f3</td>
<td></td>
</tr>
<tr>
<td>f7</td>
<td>S1.1(2) S6.2.2(2) S6.2.4(1)</td>
</tr>
<tr>
<td>f8</td>
<td>S2.1(3) S3.1(6) S6.1(3) S6.2.1(2) S6.2.3(1) S6.3(2) S7.1(4)</td>
</tr>
<tr>
<td>f6</td>
<td>S3.1(6) S5.1(2) S6.1(2) S6.2.1(2) S6.2.2(4) S6.3(2) S7.1(3)</td>
</tr>
<tr>
<td>f37</td>
<td>S2.1(3) S3.1(6) S4.1(3) S4.2(5) S6.1(2) S7.1(5)</td>
</tr>
<tr>
<td>f42</td>
<td>S2.1(3) S3.1(7) S4.1(3) S4.2(4) S4.3.1(2) S7.1(5)</td>
</tr>
<tr>
<td>f43</td>
<td>S2.1(3) S4.1(3) S4.3.1(1) S5.1(2) S5.2(2) S7.1(1)</td>
</tr>
</tbody>
</table>

END.

# III. SUBSYSTEMS SECTION

S1  {  }  

S1.1 (< p9 p33 >)  

[ ]  

{ (f3 : FH-1 ) (f4 : S6 ) (f5 : S6 ) (f7 : FH-1 ) (f12 : S6 ) (f40 : S3.1 ) }  

S2  {  }  

S2.1 (< p55 p60 > p38 )  

[ ]  


S3  { (f44< S2.1 S3.1 S3.2 S7.1 >) }  

S3.1 (< p76 p79 > p72 p62 p32 p73 p66 p77 p74 p44 )  

[ (f40< S1.1 S7.1 >) (f46< S5.1 S7.1 >) (f57< >) (f58< >) (f59< >) (f60< >) (f63< >) (f68< >) (f54< >) (f49< >) (f51< >) (f52< >) (f61< >) ]  


S3.2 (< p67 p68 >)  

[ ]  

{ (f16 : S6 ) (f41 : S4 ) (f44 : S3 ) }
S4
{ (f10< S3.1 S4.1 S4.2 S4.3.1 S4.3.2 S6.1 S6.2.1 S6.3 >)
 (f38< S2.1 S4.1 S4.2 S6.1 S7.1 >)
 (f41< S3.1 S3.2 S4.1 S4.2 S4.3.1 S4.3.2 S7.1 >) }
S4.1
(< p11 p13 > p46 )
[
]
{ (f10 : S4 ) (f37 : FH-1 ) (f38 : S4 ) (f41 : S4 )
 (f42 : FH-1 ) (f43 : FH-1 ) (f3 : FH-1 ) }
S4.2
(< p15 p49 p53 > p47 p70 )
[
]
{ (f3 : FH-1 ) (f10 : S4 ) (f41 : S4 ) (f42 : FH-1 )
 (f43 : FH-1 ) }
S4.3
{ }
S4.3.1
(< p12 p48 >)
[
]
{ (f3 : FH-1 ) (f10 : S4 ) (f41 : S4 ) (f42 : FH-1 )
 (f43 : FH-1 ) }
S4.3.2
(< p14 p52 >)
[
]
{ (f10 : S4 ) (f41 : S4 ) (f42 : FH-1 ) }
S5
{ }
S5.1
(< p43 p45 >)
[
]
{ (f3 : FH-1 ) (f6 : FH-1 ) (f9 : S6 ) (f43 : FH-1 )
 (f46 : S3.1 ) }
S5.2
(< p39 p42 >)
[
]
{ (f3 : FH-1 ) (f9 : S6 ) (f43 : FH-1 ) (f4 : S6 ) }
S6
{ (f4< S1.1 S3.1 S5.2 S6.1 S6.2.1 S6.2.2 S6.2.3 S6.2.4
 S6.3 S7.1 >)
 (f5< S1.1 S2.1 S3.1 S6.1 S6.2.1 S6.2.2 S6.2.3 S6.3 >)
 (f9< S3.1 S5.1 S5.2 S6.1 S6.2.1 S6.3 >)
 (f12< S1.1 S6.1 S6.2.1 S6.2.2 >)
 (f13< S3.1 S6.1 S6.2.1 S6.2.4 S6.3 >)
 (f15< S3.1 S6.1 S6.2.1 S6.2.2 >)
 (f16< S3.1 S3.2 S6.1 S6.2.1 >)
 (f18< S2.1 S3.1 S6.1 S6.2.1 >) (f19< S6.1 S6.2.1 >)
 (f1< S6.1 S6.2.3 S6.3 >) (f2< S6.1 S6.3 >) }
S6.1
(< p3 p36 > p82 )
[
]
{ (f39< -->) }
{ (f3 : FH-1 ) (f4 : S6 ) (f5 : S6 ) (f6 : FH-1 )
 (f7 : FH-1 ) (f8 : FH-1 ) (f9 : S6 ) (f10 : S4 )

164

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
(f12 : S6) (f13 : S6) (f15 : S6) (f16 : S6)
(f18 : S6) (f19 : S6) (f37 : FH-1) (f38 : S4)
(f1 : S6) (f2 : S6)

S6.2
{(f14 < > )}

S6.2.1
(< p2 p20 >)
[
{(f3 : FH-1) (f4 : S6) (f5 : S6) (f6 : FH-1)
(f7 : FH-1) (f8 : FH-1) (f9 : S6) (f10 : S4)
(f12 : S6) (f13 : S6) (f14 : S6.2) (f15 : S6)
(f16 : S6) (f17 : S7.1) (f18 : S6) (f19 : S6) }

S6.2.2
(< p19 p81 > p63 p61 p25 p64)
[
{(f3 : FH-1) (f5 : S6) (f6 : FH-1) (f7 : FH-1)
(f15 : S6) (f4 : S6) (f12 : S6) }

S6.2.3
(< p21 p24 >)
[
{(f1 : S6) (f3 : FH-1) (f4 : S6) (f7 : FH-1)
(f17 : S7.1) (f5 : S6) (f8 : FH-1) }

S6.2.4
(< p28 p30 > p8)
[
{(f3 : FH-1) (f4 : S6) (f13 : S6) (f14 : S6.2)
(f7 : FH-1) }

S6.3
(< p1 p59 >)
[
{(f1 : S6) (f2 : S6) (f3 : FH-1) (f4 : S6) (f5 : S6)
(f6 : FH-1) (f7 : FH-1) (f8 : FH-1) (f9 : S6)
(f10 : S4) (f13 : S6) }

S7
{}

S7.1
(< p16 p23 > p18 p22 p27)
[
{(f17 < S3.1 S6.2.1 S6.2.3 >) }
{(f3 : FH-1) (f4 : S6) (f6 : FH-1) (f7 : FH-1)
(f8 : FH-1) (f37 : FH-1) (f42 : FH-1) (f44 : S3)
(f38 : S4) (f40 : S3.1) (f41 : S4) (f43 : FH-1)
(f46 : S3.1) }

END.

*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/*/
# IV. SINGULAR PROGRAMS SECTION

p78(20) p54(11) p31(6) p84(2) p51(2) p50(2) p40(2) p10(2)

END.
# V. UNCONNECTED COMPONENTS SECTION

p83(2) p35(1) p7(1) p65(1) p56(1) p41(1) p85(1) p17(1) p57(1)  
p26(1) p29(1) p58(1) p69(1) p5(0) p4(0) p71(0) p87(0) p37(0)  
p75(0) p86(0) p88(0) p6(0) p80(0) p34(0) p89(0)

f45(1) f20(1) f52(1) f24(1) f21(1) f26(1) f64(1) f28(1) f65(1)  
f48(1) f66(1) f30(1) f67(1) f32(1) f22(1) f33(1) f69(1) f56(1)  
f70(1) f35(1) f71(1) f11(1) f72(1) f25(1) f73(1) f47(1) f74(1)  
f31(1) f75(1) f35(1) f76(1) f36(1) f77(1) f27(1) f78(1) f53(1)  
f79(1) f23(1) f80(1) f34(1) f81(1) f29(1) f82(1)

END.

# VI. SUPRA SUBSYSTEMS SECTION

S6  
S1

S3  
S7  
S5  
S4  
S2

END.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.
VITA

Carlos Montes de Oca Vázquez was born on April 13, 1966, in the city of Zacatecas, which is the capital of the state of Zacatecas in Mexico. He graduated Magna Cum Laude in Computer Engineering from the Instituto Tecnológico y de Estudios Superiores de Monterrey (ITESM), Monterrey Campus, in June 1986. In 1993, Mr. Montes de Oca obtained his master of science degree in Computer Science from the Universidad Autónoma Metropolitana (UAM-A), Azcapotzalco Campus, in Mexico City. His master’s project was considered one of the best master’s projects in the UAM-A. In the same year, he was granted a Fulbright-Conacyt scholarship to pursue doctoral studies at Louisiana State University in Baton Rouge, Louisiana, in the United States of America.

Mr. Montes de Oca has worked in both industry and academia. He was the information systems director of Sixtar, a jewelry manufacturer in Mexico City, and the head of the administrative information systems department in the Zacatecas Autonomous University. In addition, Mr. Montes de Oca has worked as researcher for Condumex, as instructor for the ITESM, Zacatecas campus, and as research assistant in the Computer Science Department at Louisiana State University. He will receive the degree of Doctor of Philosophy in Computer Science in May, 1999.
DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Carlos Montes de Oca

Major Field: Computer Science

Title of Dissertation: Design Recovery and Data Mining: A Methodology that Identifies Data Cohesive Subsystems Based on Mining Association Rules

Approved:

[Signatures of Major Professor and Chairman, Dean of the Graduate School]

EXAMINING COMMITTEE:

[Signatures of committee members]

Date of Examination: April 5, 1999