Recasting Program Reverse Engineering through On–Line Analytical Processing

by

Periklis Andritsos

A thesis submitted in conformity with the requirements
for the degree of Master of Science
Graduate Department of Computer Science
University of Toronto

Recasting Program Reverse Engineering through On-Line Analytical Processing

Periklis Andritsos

Master of Science

Department of Computer Science

University of Toronto 2000

Abstract

Program reverse engineering is the task that helps software engineers understand the architecture of large software systems. We study how the data modeling techniques known as On–Line Analytical Processing (OLAP) can be used to enhance the sophistication and range of reverse engineering tools. This is the first comprehensive examination of the similarities and differences in these tasks, both in how OLAP techniques meet (or fail to meet) the needs of reverse engineering and in how reverse engineering can be recast as data analysis.

We identify limitations in the data modeling tools of OLAP that are required in the area of reverse engineering. Specifically, multidimensional models assume that while facts may change dynamically, the structure of dimensions are relatively static (both in their dimension values and their relative orderings). We show both why this is required in current OLAP solutions and provide new solutions that effectively manage dynamic dimensions.

Contents

List of Figures

Chapter 1

Introduction

1.1 Basic Background

In recent years, an increasing number of organizations have realized the competitive advantage that can be gained from the efficient access to accurate information. Information is a key component in the decision-making process. The more information we have, the better are the chances for successful decisions. A *Data Warehouse* consolidates information from different data sources and enables the application of sophisticated query tools for faster and better analysis. Data warehouses came into existence to meet the changing demands of the enterprises as *On-Line Transaction Processing (OLTP)* systems could not cover the analytical needs of the enterprises' competitive environment. OLTP systems are mainly supported by the operational databases, while they automate daily tasks, such as the banking transactions [CD97]. Data Warehouses, on the other hand, support analytical capabilities by providing an infrastructure for integrated, company-wide, historical data from which the analysis process can be achieved [IH94]. A data warehouse integrates an enterprise, which is comprised of many, older and even incompatible application systems.

One of the basic keywords in data warehouse technology is *Dimensional Modeling* [Fir98]. In Dimensional Modeling, a set of tables and relations forms a model, whose purpose is to optimize decision support query performance, relative to a measure or set of measures of the outcome of the business process that is modeled. Using the Dimensional Modeling approach, developers first decide on the business processes that are going to be modeled and then on what each low level record in the *fact table* will represent. For example, one would have all the transactions made in a bank, on a specific date and place by all customers, stored in a table and analyze them according to the "time", "geography" and "personal information" dimensions.

The concept of data warehousing has been widely used in business-oriented applications. It enables analysts, managers and executives to gain clear insight into data of their enterprise, detect anomalies and make strategic decisions for a better position of their company among others in the market. The main characteristics of this technology are [Vas98]:

- the multidimensional view of data; and
- the data analysis which can be performed through interactive and navigational querying of data.

In a multidimensional system, there exists a set of *numeric measures* (the objects of analysis), which are viewed as points in a multidimensional space consisting of several dimensions. Each dimension can be described by a set of attributes that are related to each other according to a hierarchy. Example dimensions can be "product", "geography", "time", *etc.*, while example measure can be the "dollar amount" of products or the "revenue of employees".

In this work, our main contribution will be to bring data warehouses and reverse engineering together. In particular, we will examine how software engineers can benefit from a multidimensional view of large software and how data analysts can benefit from access to hidden structures in their data, obtained by *Reverse Engineering* approaches. The hidden structures mostly involve graphs, which we believe can be explored and browsed using data warehousing techniques.

Reverse Engineering involves two phases [Til92]:

1. the identification of the components of a system and their interdependencies; and
2. the extraction of system abstractions and design information.

Intuitively, reverse engineering helps developers understand the architecture of large legacy software systems. Several tools have been designed and built towards this goal, mainly because the majority of legacy systems are undocumented. Even if documentation exists, these systems help people compare the as-implemented with the as-documented or the as-designed structure of the underlined system. Ciao [CFKW95], Dali [KC97], ManSART [YHC97], PBS [FHK+97], Rigi [MOTU93] and SPOOL [KSRP99] are tools that have evolved as products of the reverse engineering research.

The above tools basically operate at two levels of abstraction [Til92]:

- the code-level of abstraction; and
- the architectural-level of abstraction.

At the code-level, the focus is on implementation details, such as instantiation of variables, how expressions are affected by certain variable values and which functions call other functions inside a program. On the other hand, systems that operate at the architectural-level extract facts that lead to a reconstruction of the actual system. Rigi, Dali and PBS use an entity-relationship model, at the conceptual level, to represent facts about a software system and an individual format for encoding the entities and relations. In addition, they are all 'open', in that they can be retargeted into different fact extractors or programming languages. Fact extractors are specific applications that examine the source code and reveal the interconnections between its entities. However, none of these tools uses a *Multidimensional Database* (MDDB) to store its facts.

All the aforementioned systems use graph structures as intermediate or final data structures to model intra– and inter–dependencies of the different components of a program. By intra–dependencies we usually mean any dependencies that exist among the data items of a procedure, a file, *etc.*, and by inter–dependencies the interactions between two or more procedures. Examples include *call graphs* and *Module Dependency Graphs (MDG)s.*

Understanding the intricate relationships that may exist between the source code components of a software system may become a difficult task, but at the same time it is crucial for the maintenance of the system [MMR+98]. This maintenance will have negative effect, if it is not based on subsystem knowledge. The situation is even worse in the case of huge systems, where any architectural view of them is not easy to infer from the source code. That is the case where data warehouses, in conjunction with *On–Line Analytical Processing (OLAP)* systems, can help. OLAP systems are widely used to allow the interactive analysis of data properly modeled in a multidimensional way.

1.2 Motivation

Due to the presence of hidden structures that involve graphs, the proper use and analysis of such structures is of evident value. Identifying the components of a software system and the interactions between them using graph theoretical algorithms is one side of the coin. The other side consists of applying mining algorithms to partition the modules of a program into meaningful and natural regions.

Furthermore, a closer look at a software program reveals interesting information concerning its structure. Structure that has either to do with their physical or architectural design. The reverse engineering tools we referred to in the previous section (PBS, Rigi, Dali *etc.*) basically use the physical structure of a software system under investigation to infer the architectural structure. This happens because it is often the case that documentation is non–existent for a software system (e.g. *Linux* [BHB99]).

Recently, the applicability of data mining in decision making tasks has become necessary since its results provide insightful information that reverse engineering tools are not able to reveal. Data mining algorithms, such as *Hierarchical Clustering* and *Concept Analysis*, appear to be promising as far as software systems are concerned. Wiggerts gives a substantial analysis as to why and how clustering algorithms help in the renovation and maintenance of legacy systems [Wig97]. At the same time, several authors have been involved in the analysis of systems using Concept Analysis [ST98, vDK99, SR97, MG99]. The identification of modules, program structure and other characteristics are boosted by the application of such an algorithm.

Both mining techniques mostly unveil:

- dependencies among the entities of the system;
- groupings or partitionings of entities; and
- relationships, especially hierarchical relationships, between entity groupings.

Navigation (or *browsing*) through these groups and hierarchies might help software engineers maintain or even understand the systems under consideration. In this work, we try to investigate how data warehouse technology and reverse engineering techniques can work together. In particular, we shall focus on how the multidimensional view of data helps in asking complex *ad hoc* queries over the information extracted by reverse engineering tools.

1.3 Contributions of the thesis

In this thesis our contributions are the following:

- we investigate how techniques, including data mining, can be used to partition and aggregate graphs, including program analysis graphs, using On–Line Analytical Processing techniques;
- we propose a multidimensional model for managing these groupings. Our results enhance the reverse engineering process by permitting integrated browsing and analysis of the data produced by these automated techniques, together with data produced by more human–centric documentation or reverse engineering techniques;
- we identify shortcomings in current OLAP techniques when applied to reverse engineering data. We propose OLAP extensions specifically designed to permit easy updates when the schema is modified by the introduction of new reverse engineering results. In particular, current OLAP models assume the structure and schema of groupings and hierarchies is static. Our solution relaxes this restriction;
- we conclude with an example of querying our multidimensional data.

The thesis is organized as follows.

- In **Chapter 2**, we give an overview of OLAP) and *Reverse Engineering* systems. We conclude giving our ideas of how these techniques can work together.
- In **Chapter 3**, we give the basics of hierarchical and heuristic algorithms for the partitioning of graphs and after discussing some of the limitations of the aforementioned algorithms we conclude with our arguments on how to incorporate the results of mining (in particular clustering) algorithms into an On–Line Analytical Processing framework, to enhance the reverse engineering process.
- In **Chapter 4**, we introduce our multidimensional model for the results of a Hierarchical and Concept Analysis algorithm.
- In **Chapter 5**, we give the intuition behind extending an existing data model that gives first–class status to dimensions in a data warehouse (the SQL($\mathcal{H}$) data model). We enumerate its limitations and give all definitions of our extended ESQL($\mathcal{H}$) data model and query language.
- In **Chapter 6**, we conclude and offer suggestions for further research on this area.

Chapter 2

Handling groups using *OLAP* and its use in *Reverse Engineering*

This chapter gives an overview of OLAP systems and their usage in the analysis of business data. Moreover, since groups can be identified in data emitted from *Reverse Engineering* tools, we give our ideas on how we could take advantage of OLAP systems to navigate and query such data.

2.1 What is OLAP

In an OLAP system, data are presented to the user in a multidimensional model, which comprises one or more *fact tables* and *dimensions*. A fact table consists of columns, each one corresponding to a dimension ,*e.g.*, geography, product and one (or more) corresponding to the measure (or measures), *e.g.*, sales amount. An example data warehouse containing the dimensions: `location, time, product` and the fact table `sales` is depicted in Figure 2.1

Furthermore, OLAP operations, such as *roll-up* or *drill-down*, provide the means to navigate along the dimensions of a data cube (we assume that the fact table is the relational representation of a data cube, the n–dimensional presentation of data [GBLP95]).

While OLAP systems have the ability to answer "who?" and "what?" questions, it is their ability to answer "what if?" and "why?" that sets them apart from Data Warehouses.

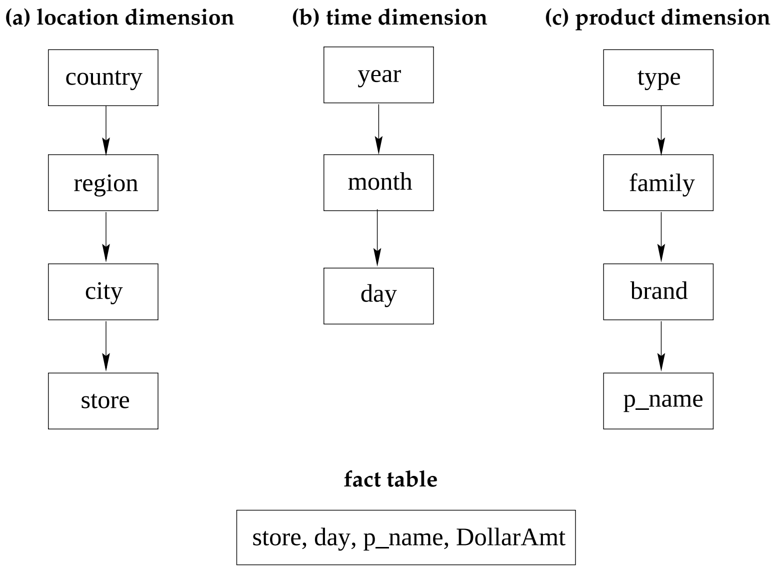

Figure 2.1: An example data warehouse

OLAP enables decision-making about future actions. A typical OLAP calculation is more complex than simply summing data, for example: "What would be the effect on suit costs if fabric prices went down by 0.20/inch and transportation costs went up by 0.10/mile?".

Based on the underlying architecture used for an OLAP application, vendors have classified their products either as **Multidimensional OLAP (MOLAP)** or **Relational OLAP (ROLAP)**.

Multidimensional OLAP uses data stored in a multidimensional database (MDDB) so as to provide OLAP analysis. As shown in Figure 2.2, MOLAP is a two-tier, client/server architecture, in which the MDDB serves as both the database layer and the application logic layer. In the database layer it is responsible for data storage, access and information retrieval while in the application logic layer takes care of all the OLAP requests. Finally, the presentation layer integrates with the application logic layer to provide an interface through which users can issue their queries.

On the other hand, Relational OLAP supports OLAP analysis, by accessing data stored in relational tables, *i.e.* a data warehouse. Figure 2.3 depicts the general architecture of a ROLAP system. It is evident that ROLAP is a three-tier, client/server architecture, in which the database uses conventional relational databases for data storage, access and

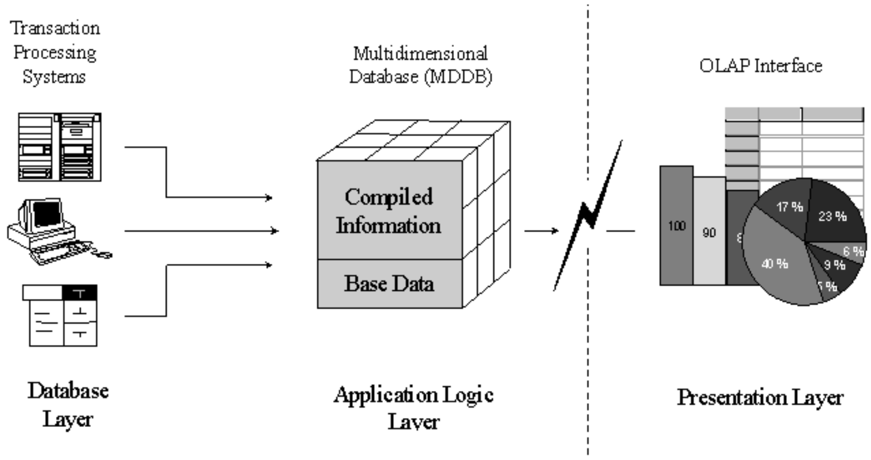

Figure 2.2: Multidimensional OLAP (MOLAP) Architecture

information retrieval. At the application logic layer, a ROLAP engine executes the multi-dimensional reports from the users and integrates with various presentation layers, through which users issue their queries.

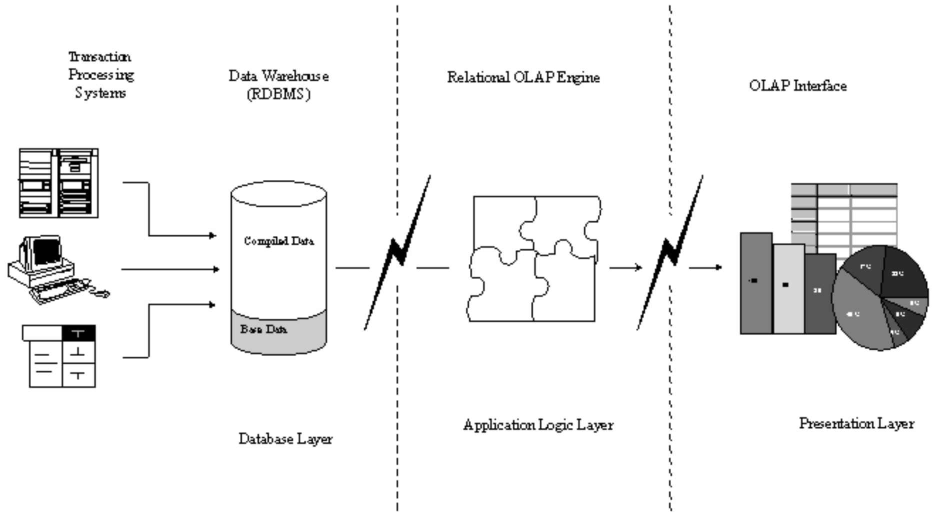

Figure 2.3: Relational OLAP (ROLAP) Architecture

Apart from the different ways that the above architectures store their data, they both provide managers with the information they need to make effective decisions about an organization's strategic directions. The key indicator of a successful OLAP application is its ability to provide information as needed, *i.e.*, its ability to provide "just-in-time" information for effective decision-making. Furthermore, due to the fact that data relationships may not be known in advance, the data model must be flexible. A truly flexible data model ensures that OLAP systems can respond to changing business requirements as needed for effective decision making.

Although OLAP applications are found in widely divergent functional areas, they all require the following key features [Cou, AP98].

- **Multidimensional view of data**, which provides more than the ability to ”slice and dice”; it gives the foundation for analytical processing through flexible access to information. Database design should not prejudice which operations can be performed on a dimension or how rapidly those operations are performed. Managers must be able to analyze data across any dimension, at any level of aggregation, with equal functionality and ease.

- **Calculation–intensive capabilities**. OLAP databases must be able to do more than simple aggregation. While aggregation along a hierarchy is important, there is more to analysis than simple data roll-ups. Examples of more complex calculations include share calculations (percentage of total) and allocations (which use hierarchies from a top-down perspective).

- **Time intelligence**. Time is an integral component of almost any analytical application. Time is a unique dimension because it is sequential in character (January always comes before February). True OLAP systems understand the sequential nature of time. At the same time business performance is almost always judged over time, for example, this month *vs.* last month, this month *vs.* the same month last year.

2.2 What is Reverse Engineering

Many systems, when they age, become difficult to understand and maintain. Sometimes, this task also becomes inefficient due to its high cost. A “Reverse engineering environment can manage the complexities of program understanding by helping the software engineers extract high–level information from low–level artifacts" [Til98].

A major effort has been undertaken in the software engineering community to produce tools that help program analysts uncover the hidden structure of legacy code. We already mentioned *Rigi* and *The Software Bookshelf* as two results of this effort [MOTU93, FHK+97]. These systems are basically focused on performing the central reverse engineering tasks presented in [Til98].

1. **Program Analysis**. This is the task where source code analysis and restructuring is performed.

2. **Plan Recognition**. This is the task where common patterns are identified. The patterns can be behavioral or structural, depending on what relationships we are looking for in the code.

3. **Concept Assignment**. This is the task that allows the software engineers to discover human–oriented patterns in the subject system. This task is still at an early research stage.

4. **Redocumentation**. This is the task that attempts to build documentation for an undocumented, and probably old, system, that describes its architecture and functionality.

From the above, it is obvious that reverse engineering tools try to extract an already existing, but unknown, structure of a software system. This involves the break down of the system either in system–oriented or human–oriented partitions that represent natural groupings, *i.e.*, different subsystems or directories of the same system.

The system examination and management is based on the use of graph structures that are produced, and later on presented to the user, taking into advantage features of the original code, such as function calls or file inclusion. In the next section, we discuss how those natural groupings could be handled by an OLAP framework.

2.3 Can OLAP be used for Reverse Engineering ?

To the best of our knowledge, program analysts have not taken advantage of a multidimensional view of data that could help them model and analyze the alternative postulated program structures. For instance, we could consider function calls stored in a fact table as in Figure 2.4. In that figure, *Function2* is called by *Function1*.

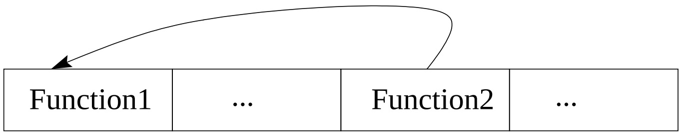

Figure 2.4: A Fact table with a graph structure

In these cases, we would like to be able to identify useful graph partitions that basically correspond to parts of the graph with a certain property. For instance, if the graph constitutes a system's Module Dependency Graph, dense regions of the graph may correspond to separate logical modules or subsystems of the system.

The issue in question now is how do we identify worthy graph partitions, having a table like the one in Figure 2.4, *i.e.*, how do we horizontally partition the fact table into subtables with a certain structure and how do we efficiently query these systems. Therefore, a particular algorithm has to be used to produce the partitions of the fact table, and in addition an OLAP system to represent and query the results. At this point, we would like to stress that our focus will not be on imposing a specific structure on our table but extracting its inherent one. This process should be based on the following decision steps:

(1) What is the current format of our data;

(2) What is the algorithm under consideration; and

(3) What models have OLAP researchers proposed and used, and how information about the results of our techniques can be incorporated in them.

Upon this, we can now answer "what if?", "why?" and "where?" questions on the system under consideration. An example OLAP query could be: "What would be the effect on the memory subsystem if the function call to `foo()` from `bar()` is omitted and the `io.h` header file is moved to the `/system` directory?". Unlike traditional OLAP, the effect will likely not be a numeric aggregate, but rather a new grouping of entities produced either by a query or by mining or program analysis algorithms. This also makes the the "multidimensional view of data" and "time intelligence" properties of more importance compared to the "Calculation–intensive capabilities" one.

Moreover, current reverse engineering tools do not support Version Control of a software system. In order to investigate differences among versions of the system, one needs to examine all versions individually, and manually find all points of interest. On the other hand, in an OLAP framework, time is treated as a separate dimension, making historical data easier to analyze.

In our work, we shall consider data that are originally in the *Rigi Standard Format (RSF)* format [MWT94, WTMS94] which is used by existing systems, such as *Rigi* and *The Software Bookshelf* [FHK+97], to provide understanding of software legacy systems. In the following chapter we present some graph theoretical and mining algorithms that can be used to unveil groups in software engineering data which can later on be modeled in an OLAP environment.

Chapter 3

Identifying Partitions and the use of Hierarchies

As already mentioned in previous chapters, the problem of analyzing and understanding information related to a software system consists of finding proper and meaningful partitions of a graph. In reverse engineering such graphs include *control flow*, *data flow* and *resource flow* graphs [MU90]. They capture the dependencies or interactions among the software entities that comprise a system.

Graph based algorithms are used in the software engineering and data mining community to find natural partitions of the set of vertices, or edges of a graph, and what follows is an overview of these techniques and how they can be used. We conclude with an analysis of which of the aforementioned techniques are suitable for adaptation in an *On–Line Analytical Processing* (*OLAP*) system.

3.1 Clustering Algorithms

3.1.1 Hierarchical Clustering

The first family of algorithms that result in groupings of the initial data set is the one of clustering algorithms. Their main purpose is to find **natural** and **meaningful** partitionings, or clusters. In some problems the produced clusters can be used "as is", while in others they may form the basis of constructing consequent clusters, thus producing a hierarchy of clusters. This section gives an overview of the two major categories of hierarchical clustering algorithms: *agglomerative* (or *bottom–up*) and *divisive* (or *top–down*). Before proceeding with the brief description of these algorithms, we introduce the notion of a sequence of partitions that are nested to each other [JD88].

Consider a set N of k data items (in our case this can be the set of nodes or edges):

$$N = \{n_1, n_2, \ldots, n_k\}$$

A partition $\mathcal{C}$ of N breaks it into subsets $\{C_1, C_2, \ldots, C_m\}$ such that:

$$C_i \cap C_j = \emptyset, 1 \leq i, j \leq m, i \neq j, \text{ and}$$

$$C_1 \cup C_2 \cup C_3 \cup \ldots \cup C_m = N$$

The set $\mathcal{C}$ is called a *clustering* and each of the C_i's a *cluster*.

Definition 1 *Partition $\mathcal{D}$ is nested into $\mathcal{C}$ if every cluster of $\mathcal{D}$ is a proper subset of a cluster of $\mathcal{C}$.*

In the following clusterings, $\mathcal{D}$ is nested in $\mathcal{C}$, but $\mathcal{D}$' is not:

$$\mathcal{C} = \{(x_1, x_3, x_7), (x_2, x_4), (x_5, x_6, x_8)\} \tag{3.1}$$

$$\mathcal{D} = \{(x_1, x_3), (x_7), (x_2, x_4), (x_5), (x_6, x_8)\} \tag{3.2}$$

$$\mathcal{D}' = \{(x_1, x_2, x_3), (x_4, x_6), (x_5, x_7, x_8)\} \tag{3.3}$$

Agglomerative Clustering

In *agglomerative* clustering [JD88], each data point starts being an individual cluster. As the algorithm goes on, clusters are merged to form larger clusters, thus nesting a clustering into another partition. The merging of clusters is based on a *similarity* (or *dissimilarity*) function that decides how similar (or dissimilar) two clusters are.

Figure 3.1 is an example of how the agglomerative function works on a data set of 4 points. A special type of tree structure is used to depict the mergings and clusterings of each level. This structure is called a *dendrogram*.

$$\{(x_1), (x_2), (x_3), (x_4)\}$$

$$\{(x_1, x_2), (x_3), (x_4)\}$$

$$\{(x_1, x_2), (x_3, x_4)\}$$

$$\{(x_1, x_2, x_3, x_4)\}$$

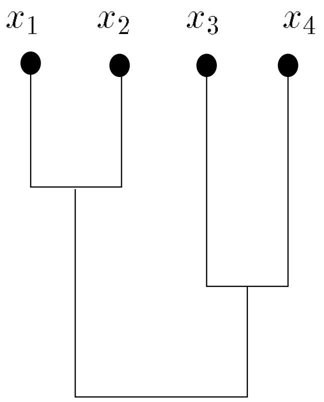

Figure 3.1: An example of agglomerative clustering

A large collection of agglomerative algorithms is presented in [JD88].

Divisive Clustering

Divisive clustering algorithms [JD88] perform the task of clustering in reverse order. Starting with all the data points in a single,"big", cluster, such an algorithm iteratively divides the "big" cluster into smaller ones. This type of algorithms are not very popular due to their high computational complexity: at each step, the number of partitions to consider is exponential [JD88].

Most of the above cases often lead to expensive solutions and maybe not near optimal ones. These are the cases where we need to employ a smart procedure to identify meaningful and interesting clusters of a graph. Heuristic approaches, then, come into play in an attempt to find optimal solutions in a moderate amount of time. Researchers use variations of already known techniques, such as hill-climbing [KL70, MMR$^+$98] to prune the search space of clusters in order to find "good" clusters in the minimum possible amount of time. Depending on the domain under consideration, different heuristics can be applied, with different results, and certain attention should be paid to their evaluation.

3.2 Discussion on the usage of graph theory and clustering algorithms in reverse engineering

Evaluating various techniques that perform clustering is crucial, and our concentration should be on the following [CW78]. three issues:

(1) on what data will the methods be applied;

(2) what is the computational cost of a method;and

(3) how "good" are the clusters.

To the above, we add a fourth issue for consideration, which emanates from the amount of disk and memory space available:

(4) whether the algorithm is suitable for in-memory execution or the data should reside on disk.

We shall be dealing with graph data from the *program domain*, specifically we'll focus on what has been called *Module Dependency Graphs* or *MDG*'s [MMR+98]. An MDG is a directed graph whose nodes are entities of a software system (procedures, files *etc.*) and whose edges are relationships between them. The nodes and edges may be accompanied with attributes that depict properties of each procedure (developer, version, fan-in, fan-out *etc.*). A software system seems easy to analyze when the number of modules (nodes) is fairly small. In this work, we are interested in analyzing large legacy systems, consisting of several thousands of nodes and edges, which often come undocumented. Our goal is to find partitionings of the MDG in a way that the produced subsets are natural and represent interesting information inherent in the system. Although there might be some structure inside a software system, we are often unable to single out individual components.

Agglomerative and divisive algorithms have been proposed by Wiggerts [Wig97] as a means of performing hierarchical clustering given an MDG-like graph. Both categories of algorithms are based on a similarity or dissimilarity measure among the nodes of the graph. This measure has to be updated each time a new clustering is formed or split into smaller ones. The measure obviously affects the number and the quality of the clusters, mainly due to the following reasons:

1. a node (module) might end up being in a wrong cluster due to ties;

2. different measures can give different clusterings.

In the hierarchical algorithms, it is not clear what happens if there are more than one edge between two nodes, hence we have a multi-graph. We would say that these algorithms are inefficient, even inapplicable in such cases. To make matters worse, parallel edges often appear in MDG's, for example when a function calls another function in two different points of the program.

Heuristic approaches seem to alleviate this pain and moreover give natural clusters of a system. Mancoridis et al. describe a system that generates meaningful clusters based on the inter- and intra-connections of nodes in MDG's [DMM99, MMCG99, MMR+98]. The clusters conform to the widely used heuristic of "low-coupling and high cohesion", a heuristic widely used in software engineering. Low coupling is a software principle which requires that interactions between subsystems should be as few as possible, while high cohesion is a related principle that requires that interactions within a subsystem should be maximized. Inside the described framework, a genetic algorithm is applied to an MDG, that explores, in a systematic way, the extremely large space of partitions and gives a "good" one. Their system, called *BUNCH*, operates well for any given set of nodes and edges.

We should note here the existence of graph theoretical algorithms that try to capture groups in graph structures. Namely, algorithms that investigate *strongly connected components* or *articulation points* might be of significant interest for our problem. Strongly connected components identify the "pieces" that comprise a graph, and two vertices are in the same component if and only if there is some path between them [Wes96]. On the other hand articulation point algorithms find vertices whose deletion disconnect a graph [Ski98]. Although both type of algorithms do not require significant amount of memory and space [CLR92], their applicability is not proven in the software reverse engineering domain.

In the previous algorithms, we could add that of finding cliques in a graph. Intuitively, a clique is a graph in which each pair of vertices is an edge. A complete graph (a graph in which all pairs of vertices form edges) has many subgraphs that are not cliques, but every induced subgraph of a complete graph is a clique [Wes96]. Finding cliques, however, is an *NP*-complete problem [CW78], and in [MM65] Moon and Moser showed that the number of cliques in a graph may grow exponentially with the number of nodes.

In our work, we do not consider any graph theoretical algorithm.An interesting question that arises when a clustering algorithm is applied, has to do with the identity of the clusters. If the algorithm is hierarchical, the question also includes the identity of the levels produced. One way to deal with it, is to use existing domain knowledge about the software system. In the following section, we present a more natural technique widely used in the software engineering community, that of *Concept Analysis*.

3.3 Concept Analysis

Concept analysis is a means to identify groupings of objects that have common attributes. In 1940 G. Birkhoff [Bir40] proved that for every binary relation between "objects" and their "attributes", a lattice can be built, which allows remarkable insight into the structure of the original relation. The following definitions and the example are taken from [LS97].

In concept analysis we consider a relation $\mathcal{T}$ between objects $\mathcal{O}$ and attributes $\mathcal{A}$, hence $\mathcal{T} \subseteq \mathcal{O} \times \mathcal{A}$. A *formal context* is the triple:

$$\mathcal{C} = (\mathcal{O}, \mathcal{A}, \mathcal{T})$$

For any set of objects $O \subseteq \mathcal{O}$, their set of common attributes is defined by:

$$\sigma(O) = \{a \in \mathcal{A} \mid \forall o \in O : (o, a) \in \mathcal{T}\}$$

while, for any set of objects $A \subseteq \mathcal{A}$, their set of common objects is given by:

$$\tau(A) = \{o \in \mathcal{O} \mid \forall a \in A : (o, a) \in \mathcal{T}\}$$

A pair (O, A) is called a **concept**, if:

$$A = \sigma(O) \text{ and } O = \tau(A)$$

Such a concept corresponds to a maximal rectangle in the table $\mathcal{T}$. A maximal rectangle is a set of objects sharing common attributes.

Concept analysis starts with the table $\mathcal{T}$ indicating the attributes of a given set of objects. It then builds up so-called *concepts* which are maximal sets of objects sharing certain features. All possible concepts can be grouped into a single lattice, the so-called *concept lattice* . The smallest concepts consist of few objects having potentially many different attributes, the largest concepts consist of many different objects that have only few attributes in common. A formal concept and its concept lattice extracted from a FORTRAN source file are shown in Figure 3.2.

```
SUBROUTINE R1(...)
COMMON /C1/ V1,V2
...
END

SUBROUTINE R2(...)
COMMON /C2/ V3,V4
COMMON /C3/ V5
...
END

SUBROUTINE R3(...)
COMMON /C2/ V3,V4
COMMON /C4/ V6,V7,V8
...
END

SUBROUTINE R4(...)
COMMON /C2/ V3,V4
COMMON /C3/ V3
COMMON /C4/ V6,V7,V8
...
END
```

	V1	V2	V3	V4	V5	V6	V7	V8
END	X	X						
END			X	X	X			
END			X	X		X	X	X
END			X	X	X	X	X	X

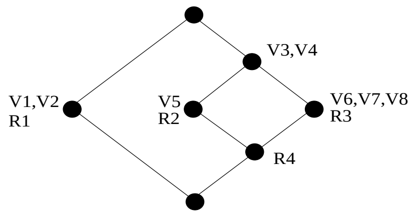

Figure 3.2: A source code, its variable usage and its concept lattice [LS97]

The set of all the concepts of a given table conform with a partial order:

$$(O_1, A_1) \leq (O_2, A_2) \iff O_1 \subseteq O_2 \iff A_1 \supseteq A_2$$

In the concept lattice, the *infimum*, or join, of two concepts is computed by intersecting their *extents*, the extent of a concept being the set of its objects O:

$$(O_1, A_1) \wedge (O_2, A_2) = (O_1 \cap O_2, \sigma(O_1 \cap O_2))$$

Thus, an infimum describes the set of attributes common to two sets of objects.
The *supremum*, or meet, is computed by intersecting the *intents*, the intent of a concept being the set of its attributes A:

$$(O_1, A_1) \vee (O_2, A_2) = (\tau(A_1 \cap A_2), A_1 \cap A_2)$$

Thus, a supremum describes a set of common objects which share the two sets of attributes.
In order to interpret a concept lattice, we also need to define the following:

$$\mu(a) = \bigvee\{c \in \mathcal{L}(\mathcal{C}) \mid a \in intent(c)\}$$

which corresponds to a lattice element labeled with a, and

$$\gamma(o) = \bigwedge\{c \in \mathcal{L}(\mathcal{C}) \mid o \in extent(c)\}$$

which corresponds to a lattice element labeled with o. The property that connects a concept lattice with its table is as follows:

$$(o, a) \epsilon \mathcal{T} \iff \gamma(o) \leq \mu(a)$$

Hence, attributes of object o are just those which show up above o in the lattice, and the objects for attribute a are those which show up below a.

Interpreting the concept lattice of Figure 3.2, we have the following, according to the aforementioned definitions:

> All subroutines below $\mu(V3)$ ($R2$, $R3$, $R4$) use $V3$ (and no other subroutines use $V3$). All variables above $\gamma(R4)$ ($V3, V4, V5, V6, V7, V8$) are used by $R4$ (and no other variables use $R4$). Thus, the concept labeled $R4$ is:
>
> $$c_1 = \gamma(R4) = (\{R4\}, \{V3, V4, V5, V6, V7, V8\})$$
>
> and the concept labeled $V5/R2$ is:
>
> $$c_2 = \mu(V5) = \gamma(R2) = (\{R2, R4\}, \{V3, V4, V5\})$$
>
> It is obvious that $c_1 \leq c_2$. This can be read as: "Any variable that is used by subroutine $R2$ is also used by $R4$". Similarly, $\mu(V5) \leq \mu(V3) = \mu(V4)$, which is read as: "All subroutines which us $V5$ will also use $V3$ and $V4$. Moreover, the infimum of $V5/R2$ and $V6, V7, V8/R3$ is labeled $R4$ meaning that $R4$ (and all subroutines below $\gamma(R4)$) uses both $V5$ and $V6, V7, V8$.

After all, the lattice uncovers a hierarchy of conceptual clusters implicit in the original table. To handle those conceptual clusters in a multidimensional way and, furthermore, query them, we need to introduce a proper multidimensional model. Several researchers have proposed variations of a multidimensional model [Fir98, IH94, PJ99, Vas98, CT97, AGS97] and our work will center towards an extension of it so as to include clusters and concepts. From our description of hierarchical clustering it seems reasonable for our task to use an agglomerative algorithm that incorporates a hierarchy, with clusters organized in a "cluster" dimension. The following chapter gives a first approach towards the multidimensional modeling of clustering and concept analysis.

Chapter 4

The Multidimensional Model

This chapter introduces our approach to the multidimensional model that will incorporate the results produced by reverse engineering tools and mining algorithms. First, we give the definitions of the model for the hierarchical clustering algorithms, and then we extend it to include the results of concept analysis.

4.1 A Multidimensional Model for managing hierarchical clusters

We consider the following.

- $\mathcal{F}$ be a set of *features* over which we perform the clustering. Hence, $\mathcal{F}$ is given by:

$$\mathcal{F} = (f_1, f_2, \ldots, f_n)$$

  where f_i can be a *function–call, a file inclusion, etc.*

- $\mathcal{A}$ be a set of *nodes*. $\mathcal{A}$ is given by:

$$\mathcal{A} = (A_1, A_2, \ldots, A_l)$$

  where A_i can be a *file, function, variable, etc.*

The dependencies (*i.e.* interactions between entities of the software system) that we have are of the form:

$$A_i \xrightarrow{f_k} A_j \tag{4.1}$$

where A_i, A_j and f_k could, for example, comply with the schema of (Figure 4.1), which is used in the *Software Bookshelf* tool [FHK+97]. Each of the nodes A_i has a *domain*, denoted by $dom(A_i)$, which corresponds to the values it can take. Intuitively, features represent

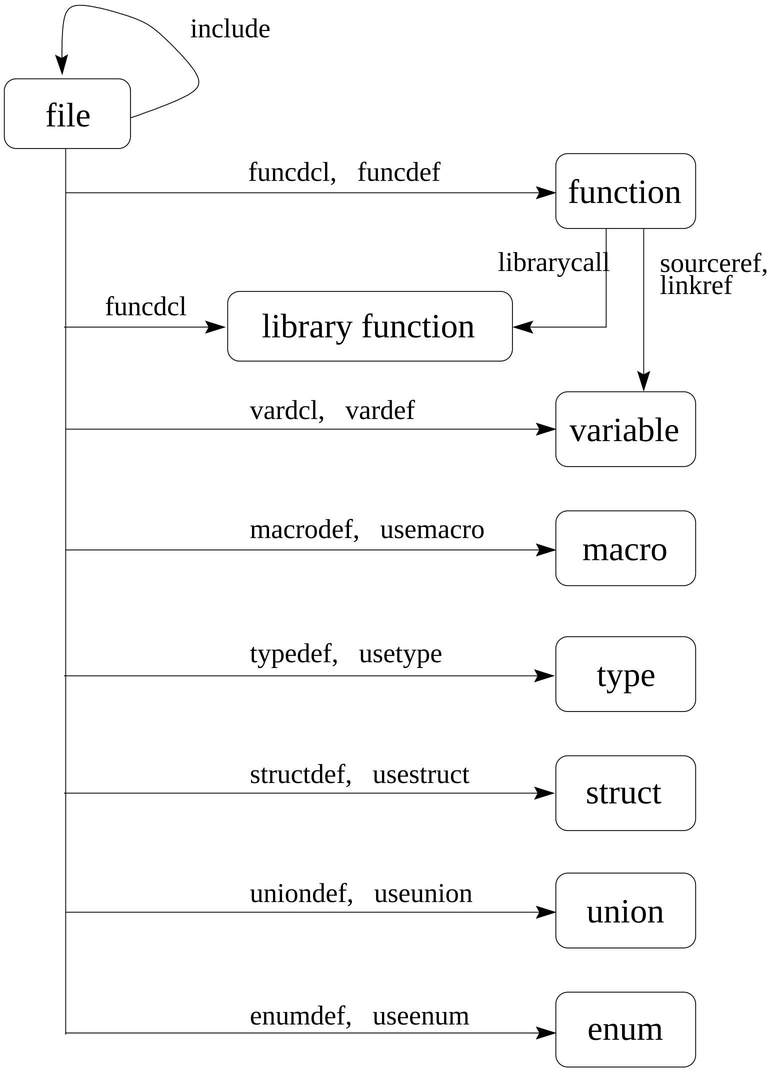

Figure 4.1: The schema for the relations used in Software Bookshelf

edge labels on the relations that connect two entities (nodes).

Each feature may be accompanied by a set of attributes: $B_1^{f_i}, B_2^{f_i}, \ldots, B_m^{f_i}$, where $B_k^{f_i}$ is the k-th attribute of feature f_i. If f_i=function-call, a set of attributes can be:

$$
\begin{aligned}
B_1^{function-call} &= \text{line number} \\
B_2^{function-call} &= \text{number of parameters passed}
\end{aligned}
$$

In the same sense, a node A_i may also be accompanied by a set of attributes: $B_1^{A_i}, B_2^{A_i}, \ldots, B_k^{A_i}$. If A_i=file, the set of attributes that can be defined is the following:

$$
\begin{aligned}
B_1^{file} &= \text{number of lines} \\
B_2^{file} &= \text{number of } \textit{if} \text{ statements} \\
B_3^{file} &= \text{number of function calls}
\end{aligned}
$$

For each feature f_i, we create a table with the following schema:

$$f_i(A_i, A_j, \{B^{f_i}\})$$

where $\{B^{f_i}\}$ is the set of attributes for f_i. In the above, f_i is a *fact table* of our Data Warehouse. Since we might have parallel edges, *i.e.*, multiple appearances of a pair (A_i, A_j), we give each pair a unique identifier, and the following is the updated schema of the fact table f_i:

$$f_i(id, A_i, A_j, \{B^{f_i}\})$$

For example, if f_i=function-call, then, according to Figure 4.1, $A_i = A_j$ =function, and a fact table could look like the following:

id	func1	func2	line #	# of parameters passed
1	foo	printf	25	0
2	foo	scanf	30	1
3	main	foo	100	3
4	main	printf	105	2
5	bar	printf	200	0

For each node A_i we create a table with the following schema:

$$p_i(A_i, \{B^{A_i}\})$$

where $\{B^{A_i}\}$ is the set of node attributes for the node A_i.

From Figure 4.1 we can easily infer that a fact table incorporates a graph structure. For instance, all dependencies of the form of Figure 4.1 constitute a graph. We are interested in splitting the nodes of the graph into meaningful horizontal partitions. Considering a node A_i, from the original set, as an individual cluster, we can apply a hierarchical clustering algorithm on that set of nodes.

An initial clustering is defined as:

$$\mathcal{D}_0^{f_i} : \{\mathcal{C}_{01}^{f_i}, \mathcal{C}_{02}^{f_i}, \ldots, \mathcal{C}_{0n}^{f_i}\}$$

where each $\mathcal{C}_{0k}^{f_i}$ is a cluster that corresponds to a node A_k, which participates in feature f_i, i.e.,

$$\mathcal{D}_0^{f_i} : \mathcal{C}_{0k}^{f_i} = A_k, \ \forall\ k$$

Given a similarity (or dissimilarity) function $\mathcal{G}$ we may start by trying to find which are the clusters that can be formed from the initial clustering $\mathcal{D}_0^{f_i}$. We call this clustering

$$\mathcal{D}_1^{f_i} = \{\mathcal{C}_{11}^{f_i}, \mathcal{C}_{12}^{f_i}, \ldots, \mathcal{C}_{1n}^{f_i}\}$$

and $\mathcal{D}_0^{f_i}$ is nested in $\mathcal{D}_1^{f_i}$ since each $\mathcal{C}_{0k}^{f_i}$ of $\mathcal{D}_0^{f_i}$ is a proper subset of a $\mathcal{C}_{1j}^{f_i}$ of $\mathcal{D}_1^{f_i}$.

Definition 2 *Operator $\lhd$ denotes the nesting of one clustering into another with respect to a feature f_i. Hence, if $\mathcal{D}_a^{f_i}$ is nested in $\mathcal{D}_b^{f_i}$, we write:*

$$\mathcal{D}_a^{f_i} \lhd \mathcal{D}_b^{f_i}$$

The cardinality, i.e. number of clusters, of a clustering $\mathcal{D}_a^{f_i}$ is denoted by $\|\mathcal{D}_a^{f_i}\|$ and thus, in the above: $1 \leq k \leq \|\mathcal{D}_a^{f_i}\|$ and $1 \leq l \leq \|\mathcal{D}_b^{f_i}\|$. Hence, $\|\mathcal{D}_a^{f_i}\| \geq \|\mathcal{D}_b^{f_i}\|$

We perform consecutive clusterings, say p, until we find the final clustering with $\|\mathcal{D}_p^{f_i}\| = 1$.

Let f_i=function–call, A_i=function with $dom(A_i) = \{foo, bar, main, printf, scanf\}$ and a similarity function $\mathcal{G}$. The clustering algorithm may give the following clusterings:

$$\begin{aligned}
\mathcal{D}_0^{function-call} &= \{(foo), (bar), (main), (printf), (scanf)\} \\
\mathcal{D}_1^{function-call} &= \{(foo, bar), (main), (printf, scanf)\} \\
\mathcal{D}_2^{function-call} &= \{(foo, bar, main), (printf, scanf)\} \\
\mathcal{D}_3^{function-call} &= \{(foo, bar, main, printf, scanf)\}
\end{aligned}$$

The schema for the above hierarchical clustering is depicted in Figure 4.2 while its instantiation in 4.3. Figures 4.2 and 4.3 represent a dimension D and its levels $\mathcal{D}_k^{f_i}$.

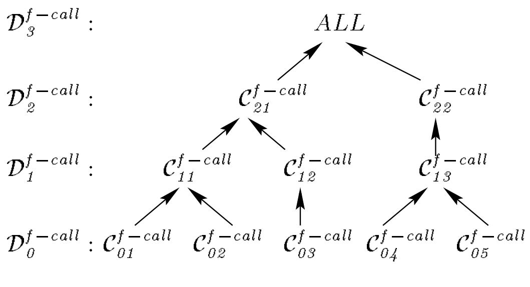

Figure 4.2: A clustering schema

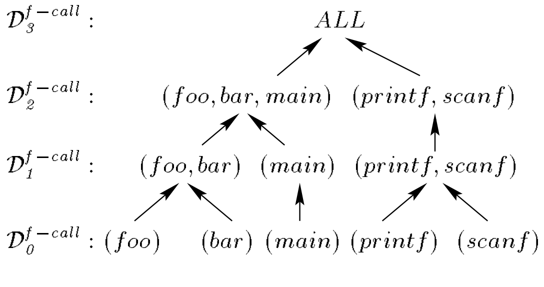

Figure 4.3: A clustering instance

Definition 3 *For each pair of clusterings $\mathcal{D}_a^{f_i}, \mathcal{D}_b^{f_i}$ such that $\mathcal{D}_a^{f_i} \lhd \mathcal{D}_b^{f_i}$, there exists a* roll–up *function $RUP_{\mathcal{D}_a^{f_i}}^{\mathcal{D}_b^{f_i}}$:*

$$\{\mathcal{C}_{bm}^{f_i} \epsilon \mathcal{D}_b^{f_i} : \exists \mathcal{C}_{ap}^{f_i}, \ldots, \mathcal{C}_{p+q}^{f_i} : \mathcal{C}_{al}^{f_i} \subseteq \mathcal{C}_{bm}^{f_i}, 1 \leq m \leq \|\mathcal{D}_b^{f_i}\|, 1 \leq p \leq l \leq p+q \leq \|\mathcal{D}_a^{f_i}\|\}$$

Intuitively, the RUP function aggregates one or more clusters of one clustering ($\mathcal{D}_a^{f_i}$) to a cluster of the immediate higher order clustering in the hierarchy ($\mathcal{D}_b^{f_i}$).

If a cluster rolls–up to another cluster with the same elements, i.e., $\mathcal{C}_{al}^{f_i} : (x_1, x_2, \ldots, x_n)$ rolls–up to $\mathcal{C}_{bm}^{f_i} : (y_1, y_2, \ldots, y_n)$ such that $x_i = x_j, \forall i, j : 1 \leq i, j \leq n$, then $\mathrm{RUP}_{\mathcal{D}_a^{f_i}}^{\mathcal{D}_b^{f_i}}$=identity.

For each pair of nodes (A_a, A_b) tha appears in the fact table f_i, i.e, the fact table that corresponds to feature f_i, we create two tables:

$$\mathcal{D}_{A_a}^{f_i}(A_a, \{\mathcal{D}^{f_i}\} \backslash \mathcal{D}_0^{f_i})$$
$$\mathcal{D}_{A_b}^{f_i}(A_b, \{\mathcal{D}^{f_i}\} \backslash \mathcal{D}_0^{f_i})$$

where $\{\mathcal{D}^{f_i}\} \backslash \mathcal{D}_0^{f_i}$ is the set of clusterings except for $\mathcal{D}_0^{f_i}$ which is represented by A_a and A_b in each table.

The dimension D^{f_i} for the pair (A_a, A_b) is given by:

$$D^{f_i} = \mathcal{D}_{A_a}^{f_i} \bigcup \mathcal{D}_{A_b}^{f_i}$$

After all, the schema of a Data Warehouse (DW) is like the one depicted in Figure 4.4 This example shows how inherent hierarchies in a software system can be revealed using a hierarchical clustering, and moreover how the results of such a clustering algorithm can be modeled in a multidimensional way. Upon the formation of such a model and storage of the data in the tables, navigation and browsing become easy and efficient.

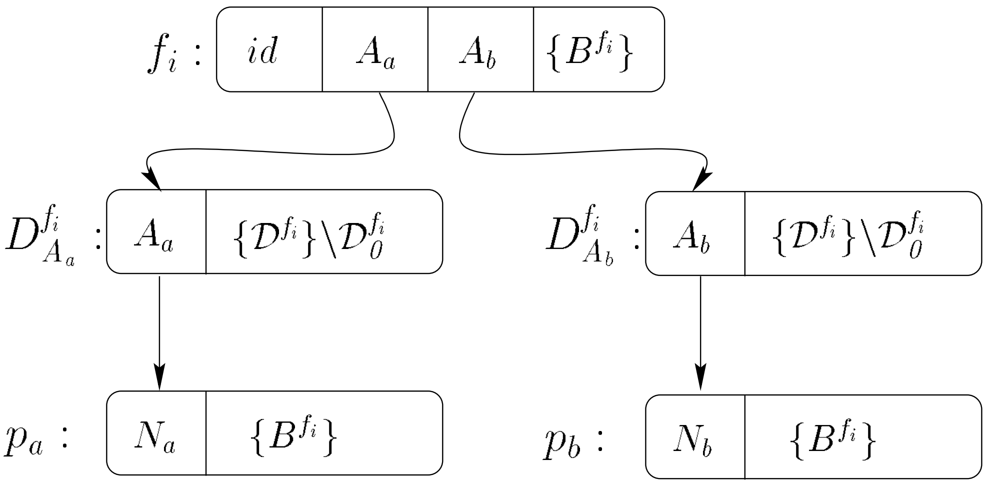

Figure 4.4: The DW schema

4.2 A Multidimensional Model for managing concepts

In our approach to hierarchical clustering, we create all groupings without actually knowing what each $\mathcal{D}^{f_i}_i$ represents. Concept Analysis is a means not only to discover the groupings but also to describe them in a more natural way.

Suppose that we have a relation $F \to A_a \times A_b$, where A_a is the set of objects and A_b is the set of attributes. For instance, n_a can be a set of nodes corresponding to `.c` files and A_b the set of nodes corresponding to `.h` files. In this example, F is the relation that depicts file inclusion. F is called the *relation matrix* in concept analysis, while here it represents the fact table, described in the previous section. An example of a relation matrix is given in Figure 4.5. The example was taken from[GMA95].

The relation matrix can be accompanied by several attributes that characterize the *edge* they represent(,*i.e.*, for a relation matrix that represents file inclusion possible attributes might be the *developer* of the `.c` files and the *number of lines* of the `.h` files).

Having such a matrix available we may start building the *concept lattice* of the above relation, which depicts maximal rectangles of a relation matrix. The concept lattice for the relation matrix of Figure 4.5 is the one of Figure 4.6.

For each object O_i that appears in a Concept C_i, we create the table `extents` with

Object	**Attribute**
1	a
1	c
1	f
1	h
2	a
2	c
2	g
2	i
3	a
3	d
3	g
3	i
4	b
4	c
4	f
4	h
5	b
5	e
5	g

Figure 4.5: Example of a *relation matrix*

the following schema:

$$extents(Object, Concept)$$

The primary key for the above table is the combination of both attributes.

For each attribute A_i that appears in a Concept C_i, we create a table `intents` with the following schema:

$$intents(Attribute, Concept)$$

The primary key for the above table is the combination of both attributes.

Finally, once the concept analysis algorithm has computed the concepts and the links between them, *i.e.*, the hierarchy inside the concept lattice, we create a table that depicts the $child \rightarrow parent$ relationships between concepts. The schema of that table is:

$$hierarchy(Concept1, Concept2)$$

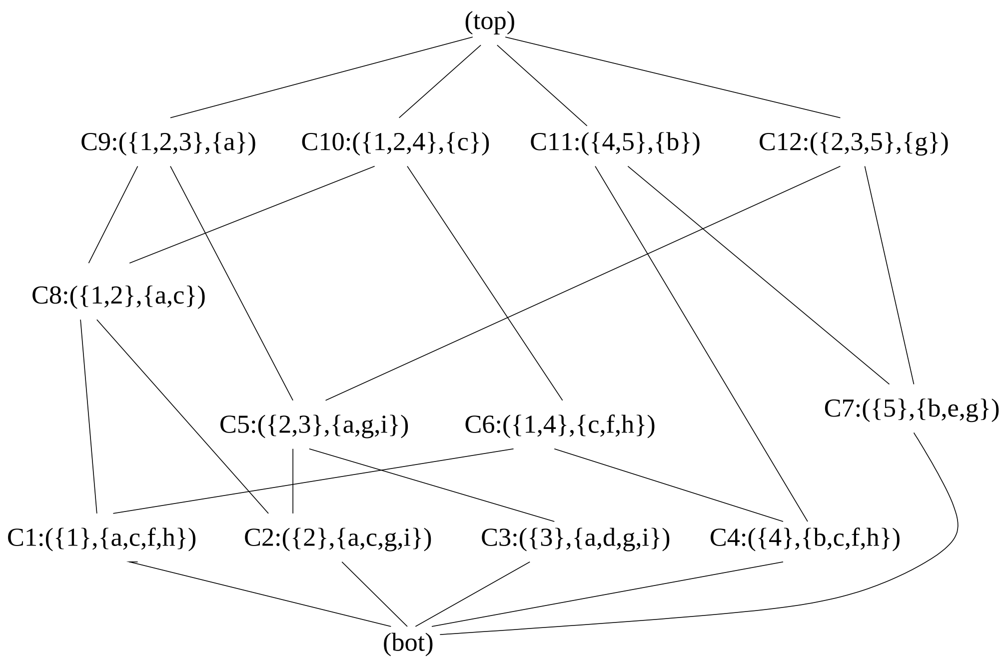

Figure 4.6: The concept lattice of the matrix in Table 1.

Primary key for the above table is the combination of both attributes. *Concept*1 is the *child* attribute, while *Concept*2 is the *parent* one.

The above tables for our example are depicted in Figures 4.7, 4.8 and 4.9.

The schema of Figure 4.4 is now extended to include the above tables. The new schema is given in Figure 4.10. Moreover, given the tables `extents, intents` and `hierarchy`, we can compute any concept in the lattice using standard SQL.

We understand that for both Hierarchical Clustering and Concept Analysis algorithms, levels are of a major importance. We need to efficiently navigate through the different levels of the hierarchy these algorithms produce and infer things that happen above or below a specific level. In general and for any instance of the Data Warehouse we need to be able to view the software system from different levels of abstraction (or detail).

The next chapter introduces the SQL($\mathcal{H}$) multidimensional model that gives first–class status to the dimensions, i.e. the hierarchies they encompass.

Object	Concept
1	c1
2	c2
3	c3
4	c4
5	c7

Figure 4.7: The `extents` table for the lattice in Figure 4.6

Attribute	Concept
a	c1
b	c4
c	c1
d	c3
e	c7
f	c1
g	c2
h	c1
i	c2
a	c2
a	c3
c	c2
c	c4
f	c4
g	c3
g	c7
h	c4
i	c3

Figure 4.8: The `intents` table for the lattice in Figure 4.6

Concept1	Concept2
bot	c1
bot	c2
bot	c3
bot	c4
bot	c7
c1	c8
c1	c6
c2	c8
c2	c5
c3	c5
c4	c6
c4	c11
c5	c9
c5	c12
c6	c10
c7	c11
c7	c12
c9	top
c10	top
c11	top
c12	top

Figure 4.9: The `hierarchy` table for the lattice in Figure 4.6

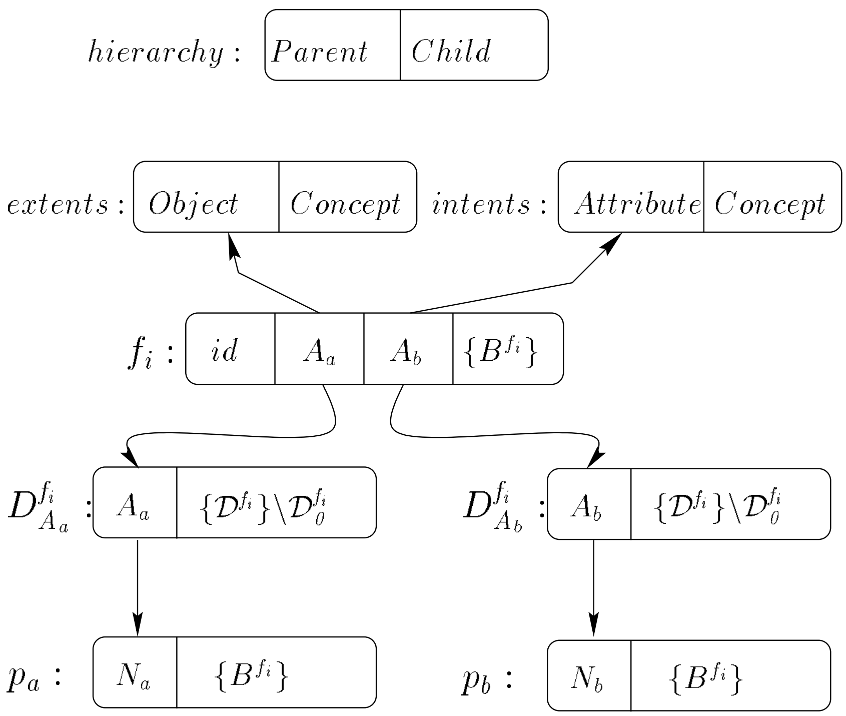

Figure 4.10: The extended DW schema

Chapter 5

The Extended SQL($\mathcal{H}$) Model

As mentioned at the end of the previous chapter, performing mining algorithms over software data needs to be flexible, in the sense that hierarchies and levels inside them should be well defined and easy to use. In the paper "What can Hierarchies do for Data Warehouses" Jagadish *et al.* proposed a new multidimensional model, called SQL($\mathcal{H}$), which extends the relational data model of SQL and gives first–class significance to the hierarchies in dimensions.

In this chapter, we briefly introduce the SQL($\mathcal{H}$) model, we identify some key weaknesses of this model and go one step further by extending it to the (E)xtended SQL($\mathcal{H}$) (or ESQL($\mathcal{H}$)) model, in order to make it more general and adaptable to our needs.

5.1 The SQL($\mathcal{H}$) model

Several models have emerged to handle multidimensional data. We can briefly mention the *Star* and *Snowflake* schemata as the most prevalent and elegant ones. However several limitations apply to these models, with heterogeneity within and across levels being one of them (especially for the Star schema). Restricting the case to Relational storage of fact and dimension tables (ROLAP architecture), those models require that the complete information concerning the levels of a hierarchy be stored in a single table. The shortcomings are straightforward. For example, having a dimension named `location`, USA and Monaco are constrained to be modeled in the same way, *e.g.* within the hierarchy `store-city-region-country`. But, as we know, Monaco is a city and a country at the same time.

The authors of [JLS99] refer to the limitations of the snowflake schema as the following:

- "Each hierarchy in a dimension has to be balanced", *i.e.* the length from the root to a leaf has to be the same.

- "All nodes at any level of a hierarchy have to be homogeneous", *i.e.*, they should include the same attributes.

Since the hierarchies in the aforementioned models are restricted to be part of the metadata, *i.e.* they do not have a first–class importance, even simple queries have to include sequences of joins making them hard to read and understand. The SQL($\mathcal{H}$) model tackles the above problem introducing an a extension of standard SQL.

The SQL($\mathcal{H}$) model comprises:

- A **Hierarchical Domain** which is a collection of attribute values arranged in such a way that form a tree. New predicates are defined over this domain, and these predicates are:

  - $=$, which is the standard equality predicate;

  - $<$, which corresponds to a binary relation over the set of attribute values so that they form a tree;

  - $<<$, which is the transitive closure of $<$; and

  - $<=$ (resp. $<<=$), which corresponds to the relation that represents non–proper child–parent (resp. descendant–ancestor) dependencies.

  In general, we interpret each hierarchical domain as a special data type.

- A **Hierarchy Schema**, which forms a rooted Directed Acyclic Graph (DAG). In this data structure the root has a special value *All*. Each node of the DAG accommodates a certain number of attributes including one that has a hierarchical domain, the *hierarchical attribute*, and is denoted by A_h.

- A **Hierarchy Instance**, which corresponds to a hierarchy schema defined as above. In the instance, all relational tables correspond to exactly one table of the schema, while at the same time no table can straddle hierarchy levels. This means that all the value a table contains belong to the same dimension. Finally, tuples of a specific table are properly related with tuples of a table (or more than one tables) above it. This means that given a tuple in a table and the hierarchy of the hierarchical attribute that corresponds to this tuple, we can infer its ancestors.

- A **Dimension Schema**, which is a name together with a hierarchy schema.

- A **Dimension Instance**, which is a name together with a hierarchy instance.

- A **Data Warehouse Schema**, which is a set of *fact tables* together with a set of dimension schemas. Fact tables are restricted to include hierarchical attributes corresponding to only the leaves of the appropriate dimension.

Imagine a Data Warehouse that includes the dimensions of `location`, `time` and `product`, and whose fact table captures dollar amounts for sales with respect to these dimensions. The schema of all tables that could form such a Warehouse are depicted in Figure 5.1.

In this figure `locId`, `tId` and `pId` are hierarchical attributes and primary keys for their respective relations.

Recall the example of the concept lattice in Figure 4.6. Trying to use the SQL($\mathcal{H}$) model to represent the schema of the concept analysis algorithm results, first of all, we observe that we do not have a tree for the hierarchical domain of the first candidate for such an attribute, which is the set of Concept Ids: $\{C_i, 1 \leq i \leq 12\}$. In order to do so, we need a more general structure. Before introducing such a structure let's see what extensions the SQL($\mathcal{H}$) model adds to standard SQL.

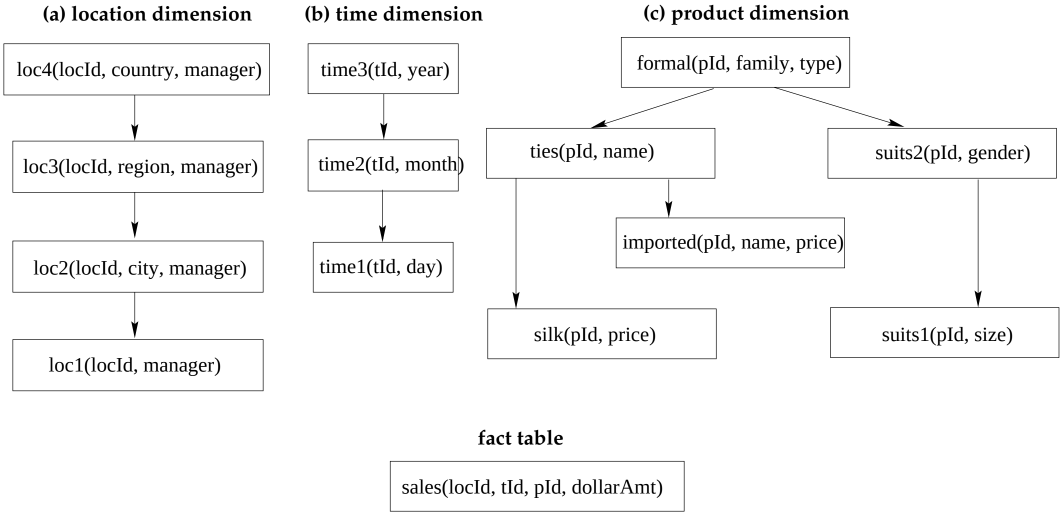

Figure 5.1: A Data Warehouse conforming to the SQL($\mathcal{H}$) model.

5.2 The Query language for the SQL($\mathcal{H}$) model

To take full advantage of the SQL($\mathcal{H}$) model a simple but powerful extension of standard SQL is proposed in [JLS99]. Considering single block SQL($\mathcal{H}$) queries, the basic extensions are the following.

- **DIMENSIONS clause**: This clause permits the inclusion of dimension names in a query. It is relevant to the tables mentioned in a FROM clause of standard SQL, but they now refer to the tables of a dimension. Moreover, just like in SQL we can declare tuple variables, in a DIMENSIONS clause all names that come right after the dimension name are called *dimension variables*. Although, it will be mentioned in the semantics of the language, dimension variables range over all tuples of all tables appearing in a dimension.

- **Hierarchical predicates**: In the SELECT, WHERE, HAVING or GROUP BY clauses of an SQL query we can include domain expressions (DEs) of the form $T.A$, where T is a tuple variable and A an attribute name. These DEs are compared with others, or values of compatible type. In order to take advantage of the hierarchical operators that are defined in the SQL($\mathcal{H}$) model, we permit DEs of the form $V.A$ where V is a dimension variable and A an attribute name. Moreover, we extend

DEs to include *hierarchical domain expressions* (HDEs) which are of the form $W.A_h$ where W is a tuple/dimension variable and A_h a hierarchical attribute. HDEs can be compared with each other using the predicates (hierarchical predicate) that are defined in the hierarchical domain. For example, given A and B, $A < B$ means that A is a child of B.

5.3 Semantics of the SQL($\mathcal{H}$) query language

For the sake of simplicity, the authors of [JLS99] use *uniform* SQL($\mathcal{H}$) queries. Such a query is of the form:

SELECT	domExpList, aggList
DIMENSIONS	dimList
FROM	fromList
WHERE	whereConditions
GROUP BY	groupbyList
HAVING	haveConditions

Clauses that also appear in standard SQL have the same semantics. The question is what happens with the newly introduced DIMENSIONS clause and the appearance of hierarchical predicates in the WHERE clause. As far as the dimension variables of the DIMENSIONS clause are concerned, "they should range over the set of nodes in the hierarchy associated with the dimension" [JLS99], *i.e.*, over all heterogeneous tuples of a hierarchy instance. The result of an SQL($\mathcal{H}$) query is a table according to the schema imposed by the sets `domExpList` and `aggList` of the SELECT clause.

The semantics of an SQL($\mathcal{H}$) query is given in the original paper [JLS99]. We present the semantics more formally in a following section, where we discuss the semantics of ESQL($\mathcal{H}$).

5.4 Limitations of the model

The model we just described offers the advantages listed below.

- Adds semantics of hierarchies to the data model and the query language:
  - gives first–class status to the hierarchies by:
    - permitting heterogeneity in dimensions, and
    - introducing hierarchical domains (trees) as first–class objects.
- Permits "dimension independence" of the queries. The DIMENSIONS clause allows the definition of dimension variables and, furthermore, allows these variables to range over the tuples of the dimension, without taking into account the schema of each table. Therefore, the evaluation of an SQL($\mathcal{H}$) query is the same no matter what is the schema of the dimension tables it refers to.
- Allows the fast evaluation of SQL($\mathcal{H}$) hierarchical queries, based on bitmap indices.

However, there exist limitations that are particularly relevant to reverse engineering data. As we already mentioned, the concept lattice of Figure 4.6 and the hierarchies that exist in it cannot be represented by the SQL($\mathcal{H}$) model. The basic restriction is that the hierarchical domain must be a tree. Another point is that if some of the hierarchies in the lattice change in time, those changes might be difficult to capture. The following list gives the two basic limitations of the model as well as the intuition behind its extension.

- Hierarchical attributes should conform to a domain that has the structure of a tree. The example of the concept lattice of Figure 4.6 proves why such a domain becomes inappropriate for mining and reverse engineering applications.
- Each level inside a hierarchy must be modeled as a separate set of tables. This implies that changes in dimension values (*e.g.*, changes in the number of levels) may lead to schema changes.

5.5 The ESQL($\mathcal{H}$) model

To overcome the limitations listed in the previous section we need to provide an extended model that fits our needs. The key point for this model is to be more general than the SQL($\mathcal{H}$) model. The notation used in the following sections is the same as in the paper of SQL($\mathcal{H}$). Definitions that are also the same are mentioned to be so. Briefly, in our model we propose:

- A more general structure for the hierarchical domain, and
- Levels to straddle tables, so that any arbitrary table may contain values from many levels.

Definition 4 [Hierarchical Domain]
A hierarchical domain *is a partially ordered set* $< \mathcal{V}_{\mathcal{H}}, \leq >$ *where* $\mathcal{V}_{\mathcal{H}}$ *is a non-empty set of attributes and* $\leq$ *a binary relation which is reflexive, antisymmetric and transitive.*

The following hold:

1. The only predicates defined on this domain are: $=, <, <=, <<, <<=$, ($<=$ is the same as $\leq$ in the above definition).
2. The equality predicate $=$ has the standard interpretation of syntactic identity.
3. The predicate $<$ is interpreted as a binary relation over $\mathcal{V}_{\mathcal{H}}$ such that for every $x, y \in L, x < y \Longleftrightarrow x \leq y \wedge x \neq y$. the graph $G_<$ over the nodes of $\mathcal{V}_{\mathcal{H}}$ can be depicted as a Hasse diagram [TM75]. Such a diagram is an undirected graph were all edges are considered as arrows from bottom to top, *i.e.*, smaller elements are placed lower.
4. The predicate $<<$ is interpreted as the transitive closure of $<$.
5. For any two elements $u, v \in \mathcal{V}_{\mathcal{H}}$, $u \leq v$ holds iff either $u < v$ or $u = v$. Respectively for $u <<= v$.

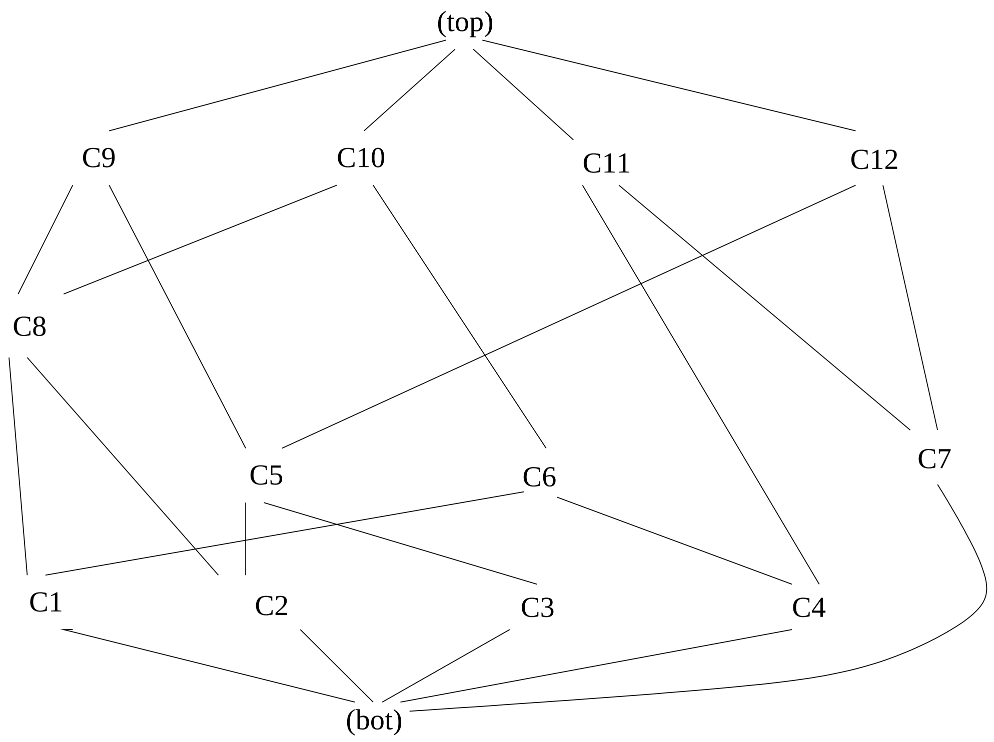

Figure 5.2: The hierarchical domain for concept ids (Cids) of figure 4.6.

The partial order of concept ids for the example of Figure 4.6 is given in Figure 5.2. Intuitively, $\mathcal{V}_{\mathcal{H}}$ is an abstract data type that corresponds to hierarchies where predicate $<$ refers to child–parent relationships and $<<$ to (proper) ancestor–descendant ones.

Whenever an attribute A conforms to a domain which is hierarchical, we call A a *hierarchical attribute* and denote it by A_h.

Definition 5 [Hierarchical Schema]

A hierarchy schema *is a triple* $\mathbf{H} = (G, \mathcal{A}, \sigma)$ *such that:*

(i) G *is a collection of nodes of any structure, having a special node All;*

(ii) $\mathcal{A}$ *is an attribute set that contains a unique hierarchical attribute* A_h*; and*

(iii) $\sigma : G \rightarrow 2^{\mathcal{A}}$ *is a function that associates a node* $u \in G$ *with a set of attributes* $\sigma(u) \subseteq \mathcal{A}$*, such that* $\forall u \neq All, A_h \in \sigma(u)$*, and* $\sigma(All) = \emptyset$*.*

All nodes of G*, except All should include the hierarchical attribute* A_h *in their attribute list.*

Imagine that we have a dimension called `Concepts` and a hierarchical attribute $A_h = Cid$ that corresponds to Figure 5.2. If the attribute set of the hierarchy is {`Cid, Objects, Attributes`}, then this attribute set can be associated with exactly one node of the hierarchy. Hence, we shall have nodes: {`Cid, Objects, Attributes`} and {} for the node *All*.

Definition 6 [Hierarchy Instance]
A hierarchy instance *corresponding to a hierarchy schema* $\mathbf{H} = (G, \mathcal{A}, \sigma)$ *is a collection of tables* $\mathcal{H}$*, that satisfy the following: each table* $r \in \mathcal{H}$ *corresponds to a unique node* $u \in G$*, (except for node All), and* r *is a table over* $\sigma(u)$*.*

Note that we do not restrict the nodes to form a DAG and we permit the straddling of tables through the levels of the hierarchy.

Definition 7 [Dimension] *[JLS99]*
A dimension schema $D(\mathbf{H})$ *is a name* D *together with a hierarchy schema* $\mathbf{H} = (G, \mathcal{A}, \sigma)$*. We refer to attributes* $\mathcal{A}$ *as the attribute set associated with dimension* D*.*

A dimension instance $D(\mathcal{H})$ *over a dimension schema* $D(\mathbf{H})$ *is a dimension name* D *with a hierarchy instance* $\mathcal{H}$ *of* $\mathbf{H}$*.*

Definition 8 [Data Warehouse Schema] *[JLS99]*
A Data Warehouse Schema *in the ESQL(*$\mathcal{H}$*) model is defined as a set of dimension schemas* $D_i(\mathbf{H}_i)$*, with associated hierarchical attributes* A_h^i*,* $1 \leq i \leq k$*, together with a set of fact table schemas of the form* $f(A_h^{j_1}, \ldots, A_h^{j_n}, B_1, \ldots, B_m)$*, where* $D_{j_1}, \ldots, D_{j_n}$ *are a subset of the dimensions* $D_1, \ldots, D_k$*, and* B_j*,* $1 \leq j \leq m$*, are additional attributes, including any measure attributes.*

A Data Warehouse, *i.e.*, a fact table and a dimension for a concept analysis framework are depicted in Figure 5.3. Note that to store the `Objects` and `Attributes` columns of the tables appearing in that figure, we are taking advantage of the object–oriented features of SQL(3) [Ram97], which permits set–valued attributes.

Dimension "Concepts"

Cid	Objects	Attributes
C1	{1}	{a,c,f,h}
C2	{2}	{a,c,g,i}
C3	{3}	{a,d,g,i}
C4	{4}	{b,c,f,h}
C5	{2,3}	{a,g,i}
C6	{1,4}	{c,f,h}
C7	{7}	{b,e,g}
C8	{1,2}	{a,c}
C9	{1,2,3}	{a}
C10	{1,2,4}	{c}
C11	{4,5}	{b}
C12	{2,3,5}	{g}

Fact Table "Basic_Concepts"

Cid
C1
C2
C3
C4
C7

Figure 5.3: An example Data Warehouse

5.6 The ESQL($\mathcal{H}$) query language

Before giving some example queries to the data model we just described, we will try to analyze the semantics of the ESQL($\mathcal{H}$) query language. The language does not have any differences with the SQL($\mathcal{H}$) query language as far as syntax is concerned. The difference is that when we try to evaluate each query, we have to take into account the new, more general hierarchical domain and the arbitrary number of tables that make up each dimension.

A *uniform* ESQL($\mathcal{H}$) query is defined in the same way as in the SQL($\mathcal{H}$) model [JLS99]. In general, a uniform ESQL($\mathcal{H}$) $\mathcal{Q}$ is a function:

$$\mathcal{Q} : \mathcal{D} \to \mathcal{R}$$

where $\mathcal{Q}$ is the set of database, and $\mathcal{R}$ a set of tables of the output, under the schema of the attribute list that appears in the SELECT clause. Taking into account all clauses of a uniform of an ESQL($\mathcal{H}$) query we have the following:

- **SELECT clause**: This clause enforces the schema of the output table. It is interpreted as in standard SQL with the addition that it may contain hierarchical and dimension attributes, *i.e.*, attributes from the set $\mathcal{A}$ of a hierarchy instance.

- **DIMENSIONS clause**: This clause permits the declaration of dimension names of interest. All dimension variables range over the set of all (possibly non–homogeneous) tuples of all tables associated with that dimension.

- **FROM clause**: This clause is interpreted exactly as in standard SQL, *i.e.*, it takes the cross product of all tables appearing in it, while all tuple variables declared in it range over all (homogeneous) tuples of the fact tables they refer.

- **WHERE clause**: To pin down the semantics of this clause, we should recall the definition of an *instantiation* function [JLS99]. Considering all tuple and dimension variables, the *instantiation* function maps them to appropriate tables of the data warehouse. Now, the key issue is to properly evaluate each `whereCond` of the WHERE clause of an the ESQL($\mathcal{H}$) query, according to the type of relationship between the operators and the operands. Thus:

  – if the `whereCond` involves attributes from the fact table and operands of the same type, the WHERE clause is evaluated exactly as in SQL. This means that all tuple satisfying the `whereCond` will appear in the result.

  – if the `whereCond` involves attributes from dimension tables which are compared to operands of the proper type based on a standard comparison operator, the query is again satisfied by all tuples in the dimension tables which appear in relationship with the operand.

  – if the `whereCond` involves hierarchical attributes which are compared to operands of the proper type based on a hierarchical predicate, the query is satisfied by all tuples that are related to operands according to the hierarchical relationship, *i.e.* the hierarchical domain of the hierarchical attribute.

In all the above, all comparisons are performed through the mapping of the instantiation function to the appropriate tuples. Taking the concatenation of all the results (instantiations) from a query and restricting those tuples to the attributes that appear in the SELECT clause, we have the final answer to the ESQL($\mathcal{H}$) query. If there is no measure defined in the fact table of the data warehouse (as in our example) instead of concatenation standard relational union should be employed.

- **GROUP BY clause**: It is interpreted exactly as in standard SQL.
- **HAVING clause**: It is interpreted exactly as in standard SQL.

5.7 Sample queries

In order to show the simplicity and power of the ESQL($\mathcal{H}$) language, we give some example queries and explain their semantics and their step by step computation.

5.7.1 Dimensional Selection

The following single block ESQL($\mathcal{H}$) query, Q1, captures the query "find concepts that contain more than 3 attributes".

```
SELECT      C.Cid
DIMENSIONS  Concepts C
WHERE       COUNT(C.Attributes) > 3
```

Here the approach is the same as in SQL($\mathcal{H}$). `C` will range over all tuples of the `Concepts` table (here the tuples are homogeneous) and select those `Cid`s that satisfy the condition of the WHERE clause. The resulting table is the following:

Cid
C1
C2
C3
C4

5.7.2 Hierarchical Join/Aggregation

The following single block ESQL($\mathcal{H}$) query, Q2, captures the query "Find the objects of each concept that contain over 2 objects".

```
SELECT      C.Cid, C.Objects
DIMENSIONS  Concepts C
FROM        Basic_Concepts F
WHERE       F.Cid <<= C.Cid
GROUP BY    C.Cid
HAVING      COUNT(C.Attributes) > 2
```

In this case the WHERE clause contains condition: "`F.Cid <<= C.Cid`", which is of the form "$W.A_h\ \theta_h$ opnd". Here, we do not have the hierarchy of the levels in the concept lattice given by the tables of the dimension `Concepts`. Thus, we should use the attributes of the hierarchical domain to get the θ_h-relation. Let's see how the query will be evaluated.

1. $\iota(C)[Cid]$ ranges over all `Cid` attributes of the fact table sales, i.e., attributes {`C1,C2,C3,C4,C7`}. For each of these attributes we compute the (reflexive) transitive closure relation tc, and get:

   - $tc(C1)$={C6,C10,C8,C9,top}
   - $tc(C2)$={C5,C9,C12,top}
   - $tc(C3)$={C5,C9,C12,top}
   - $tc(C4)$={C6,C10,C11,top}
   - $tc(C7)$={C11,C12,top}

   Now, i(opnd) ranges over all hierarchical attributes of table `Concepts`. Taking also into account the HAVING clause condition, we have:

   - Using $tc(C1)$, the instantiation of θ_h-relatives of `C1` is:

Cid	**Objects**
C10	{1,2,4}
C9	{1,2,3}
top	{1,2,3,4,5}

- Using $tc(C2)$, the instantiation of θ_h-relatives of `C2` is:

Cid	**Objects**
C9	{1,2,3}
C12	{2,3,5}
top	{1,2,3,4,5}

- Using $tc(C3)$, the instantiation of θ_h-relatives of `C3` is:

Cid	**Objects**
C9	{1,2,3}
C12	{2,3,5}
top	{1,2,3,4,5}

- Using $tc(C4)$, the instantiation of θ_h-relatives of `C4` is:

Cid	**Objects**
C10	{1,2,4}
top	{1,2,3,4,5}

- Using $tc(C7)$, the instantiation of θ_h-relatives of `C7` is:

Cid	**Objects**
C12	{2,3,5}
top	{1,2,3,4,5}

2. The final result for Q2 is the union of all the above tables:

Cid	**Objects**
C9	{1,2,3}
C10	{1,2,4}
C12	{2,3,5}
top	{1,2,3,4,5}

5.7.3 Hierarchical Join

The following ESQL($\mathcal{H}$) query, Q3, captures the query "Find the immediate breakdown of concepts with more than 2 objects".

```
SELECT      C1.Cid AS Concept1, C2.Cid AS Concept2, C2.Objects C2Objects
DIMENSIONS  Concepts C1, C2
FROM        Basic_Concepts F
WHERE       F.Cid <<= C1.Cid AND
            C2.Cid < C1.Cid AND
            C1.Cid IN ( SELECT      C.Cid
                        DIMENSIONS  Concepts C
                        FROM        Basic_Concepts F
                        WHERE       F.Cid <<= C.Cid
                        GROUP BY    C.Cid
                        HAVING      COUNT(C.Objects) > 2 )
GROUP BY    C1.Cid, C2.Cid
```

Taking into consideration the result of query Q2, it is easy to infer what the result of Q3 will be: For all tuples in the result of Q2, give the `Cid` of its immediate child. The result is given in the following table:

Concept1	**Concept2**	**C2Objects**
C9	C8	{1,2}
C9	C5	{2,3}
C10	C8	{1,2}
C10	C6	{1,4}
C12	C5	{2,3}
C12	C7	{5}
top	C9	{1,2,3}
top	C10	{1,2,4}
top	C11	{4,5}
top	C12	{2,3,5}

Chapter 6

Conclusions

In this work, we studied ways of putting *Reverse Engineering* and *Data Warehousing* techniques together. Software reverse engineering techniques try to capture the structure of, usually, undocumented systems so that their understanding and maintenance become easier. On the other hand Data Warehousing, and specifically On–Line Analytical Processing systems, provide the appropriate means to pose complex, *ad hoc*, queries on information extracted by reverse engineering tools.

We first investigated how several graph–theoretical algorithms can be used in order to analyze and partition graph structures that are extracted from reverse engineering tools, such as *Rigi* and the *The Software Bookshelf*. Most of these algorithms proved to be inefficient to implement due to time and space constraints and the nature of the graphs that appear in the results. Most important is the fact that those algorithms do not reveal any hierarchical structure of the underlying system.In the following chapters, we described how On–Line Analytical Processing systems handle situations where hierarchies exist. A large number of researchers have been involved in the study of such systems so as to make their modeling and querying easier for the naive user. These systems are basically employed by decision makers who search for trends and future estimates about their company's critical parameters. To the best of our knowledge OLAP systems have never been comprehensively studied and employed in the field of reverse engineering.

This thesis presented a new multidimensional model for *hierarchical clustering* and *concept analysis* algorithms. Both types of algorithms are often used by software engineers and their results yield interesting observations about the systems under consideration. However, they have never been able to store these results in a natural and easy to use manner. Our model is the basis for optimal storage and natural way of querying this data. Furthermore, we extended the work by Jagadish, Lakshmanan and Srivastava [JLS99], in order to give a more general multidimensional model which provides first–class status to dimensions. The basic intuition is that the algorithms mentioned above may give different results under different parameters, or given different versions of the same program. The extensions comprise:

- A more general structure for the hierarchical domain of a certain type of attributes, called hierarchical attributes; and
- A refined definition of the notion of levels in this model, so that tuples may appear in any table of a hierarchy.

Therefore, the hierarchy of levels can be extracted by the hierarchical domain of the hierarchical attributes and if new levels appear in the conceptual level, the hierarchy schema does not need to be changed.

The work presented in this thesis can be extended in several ways. We focus on the evaluation of complex OLAP queries posed over the ESQL($\mathcal{H}$) model. In [JLS99], a new algorithm based on bitmap indices is given in order to compute queries that include the $<<=$ and $=$ hierarchical predicates. This algorithm does not need to be further extended for ESQL($\mathcal{H}$) queries because it is based on a preorder traversal of the hierarchical domain. In ESQL($\mathcal{H}$) the hierarchical domain is a partial order where such a traversal can be defined. However, we need to consider algorithms for evaluating queries including the $<$ hierarchical predicate. Bitmap indices could help, and moreover, they can provide the appropriate background for the faster evaluation of queries that entail COUNT and SUM aggregate functions in their SELECT or HAVING clauses.

Bibliography

[AGS97] Rakesh Agrawal, A. Gupta, and Sunita Sarawagi. Modeling Multidimensional Databases. In Alex Gray and Per-Åke Larson, editors, *Proc. of the 13th Int'l Conf. on Data Engineering, (ICDE)*, pages 232–243. IEEE Press, 7–11 April 1997.

[AP98] Periklis Andritsos and Athanassia Papagianni. *On the development of a tool that supports OLAP queries.* Diploma thesis, Dept. of Electrical and Computer Engineering National Technical Univeristy of Athens, 1998.

[BHB99] Ivan T. Bowman, Richard C. Holt, and Neil V. Brewster. Linux as a Case Study: Its Extracted Software Architecture. In *Proc. of the 21st Int'l Conf. on Software Engineering*, pages 555–563, Los Angeles, CA, USA, May 1999. ACM Press.

[Bir40] Garrett Birkhoff. *Lattice Theory.* AMS Colloquium Public., 25. AMS, New York, 1940.

[CD97] S. Chaudhuri and U. Dayal. An overview of Data Warehousing and OLAP technology. *SIGMOD Record*, 26(1): 65–74, March 1997.

[CFKW95] Yih-Farn Chen, Glenn S. Fowler, Eleftherios Koutsofios, and Ryan S. Wallach. Ciao: A Graphical Navigator for Software and Document Repositories. In *IEEE Proc. of the Int'l Conf. on Software Maintenance*, pages 66–75, Nice, France, October 1995.

[CLR92] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to algorithms.* MIT Press and McGraw-Hill Book Company, 6th edition, 1992.

[Cou] OLAP Council. OLAP Council's White Paper. In *http://www.olapcouncil.org/.*

[CT97] Luca Cabbibo and Riccardo Torlone. Querying Multidimensional Databases. In *Proc. of the 6th Int'l Workshop on Database Programming Languages, (DBLP)*, pages 319–335, Estes Park, Colorado, 18–20 August 1997.

[CW78] D. G. Corneil and M. E. Woodward. A comparison and evaluation of graph theoretical clustering techniques. *INFOR*, 16(1):74–89, February 1978.

[DMM99] D. Doval, S. Mancoridis, and B. S. Mitchell. Automatic clustering of software systems using a genetic algorithm. In *Proc. of the Int'l Conf. on Software Tools and Engineering Practice*, Pittsburgh, PA, August 1999.

[FHK+97] P. J. Finnigan, R. C. Holt, I. Kalas, S. Kerr, K. Kontogiannis, H. A. Müller, J. Mylopoulos, S. G. Perelgut, M. Stanley, and K. Wong. The Software Bookshelf. *IBM Systems Journal*, 36(4): 564–593, 1997.

[Fir98] Joseph M. Firestone. Dimensional Modeling and E-R Modeling In The Data Warehouse. Executive Information Systems Inc., White Paper 8, June 1998.

[GBLP95] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Technical Report MSR-TR-95-22, Microsoft Research, Advanced Technology Division, Redmond, WA 98052, U.S.A., 15 November 1995.

[GMA95] Robert Godin, Rokia Missaoui, and Hassan Alaoui. Incremental concept formation aalgorithms based on galois (concept) lattices. *Computational Intelligence*, 11(2):246–267, November 1995.

[IH94] W. H. Inmon and R. D. Hackathorn. *Using the Data Warehouse.* John Wiley & Sons, Inc., 1994.

[JD88] A. K. Jain and R. C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, Englewood Cliffs, NJ, 1988.

[JLS99] H. V. Jagadish, Laks V. S. Lakshmanan, and Divesh Srivastava. What can Hierarchies do for Data Warehouses? In *Proc. of the 25th Int'l Conf. on Very Large Data Bases, (VLDB)*, pages 530–541, Edinburgh, Scotland, UK, 7–10 September 1999.

[KC97] Rick Kazman and S. Jeromy Carrière. Playing Detective: Reconstructing Software Architecture from Available Evidence. Technical Report, CMU/SEI-97-TR-010, Software Engineering Institute–Carnegie Mellon University, Pittsburg, PA 15213, October 1997.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic for partitioning graphs. *Bell Systems Technical J.*, 49:291–307, 1970.

[KSRP99] Rudolf K. Keller, Reinhard Schauer, Sebastien Robitaille, and Patrick Page. Pattern-Based Reverse-Engineering of Design Components. In *Proc. of the 21st Int'l Conf. on Software Engineering*, pages 226–235, Los Angeles, CA, USA, May 1999. ACM Press.

[LS97] C. Lindig and G. Snelting. Assessing modular structure of legacy code based on mathematical concept analysis. In *Proc. of the Int'l Conf. on Software Engineering*, Boston, MA, 17–23 May 1997. IEEE Computer Society Press.

[MG99] Renée J. Miller and Ashish Gujarathi. Mining for Program Structure. *Int'l Journal on Software Engineering and Knowledge Discovery*, ?(?):???–???, 1999.

[MM65] J. W. Moon and L. Moser. On cliques in graphs. *Israel Journal of Mathematics*, 3:23–28, 1965.

[MMCG99] S. Mancoridis, B. S. Mitchell, Y. Chen, and E. R. Gansner. Bunch: A clustering tool for the recovery and maintenance of software system structures. In *Proc.*

of the Int'l Conf. on Software Maintenance, pages 50–59, Oxford, UK, August 1999. IEEE Computer Society Press.

[MMR+98] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner. Using automatic clustering to produce high-level system organizations of source code. In *Proc. of the Int'l Workshop on Program Understanding*, Ischia, Italy, June 1998.

[MOTU93] Hausi A. Müller, Mehmet A. Orgun, Scott R. Tilley, and James S. Uhl. A reverse engineering approach to subsystem structure identification. *Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.

[MU90] Hausi A. Müller and James S. Uhl. Composing Subsystem Structures using (k,2)-partite Graphs. Technical Report DCS-128-IR, Department of Computer Science, University of Victoria, March 1990.

[MWT94] Hausi A. Müller, Kenny Wong, and Scott R. Tilley. Understanding Software Systems Using Reverse Engineering Technology. In *Proc. of the 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Sciences, (ACFAS)*, pages 41–48, Montreal, PQ, 16–17 May 1994.

[PJ99] Torben Bach Pedersen and Christian S. Jensen. Multidimensional Data Modeling for Complex Data. In *Proc. of the 15th Int'l Conference on Data Engineering, (ICDE)*, pages 336–345, 23–26 March 1999.

[Ram97] Raghu Ramakrishnan. *Database Management Systems*. McGraw-Hill, 1997.

[Ski98] Steven S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, Berlin, Germany / Heidelberg, Germany / London, UK / etc., 1998.

[SR97] Michael Siff and Thomas Reps. Identifying Modules via Concept Analysis. In *Proc. of. the Int'l Conf. on Software Maintenance*, pages 170–179, Bari, Italy, September 1997.

[ST98] Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. *ACM SIGSOFT Software Engineering Notes*, 23(6):99–110, November 1998. Proc. of the Int'l Symposium on the Foundations of Software Engineering.

[Til92] Scott R. Tilley. Management Decision Support Through Reverse Engineering Technology. In *Proc. of CASCON'92*, pages 319–328, 9–11 November 1992.

[Til98] Scott Tilley. A Reverse–Engineering Environment Framework. Technical Report, CMU/SEI-98-TR-005, Software Engineering Institute–Carnegie Mellon University, Pittsburg, PA 15213, April 1998.

[TM75] J. P. Tremblay and R. Manohar. *Discrete Mathematical Structures with Applications to Computer Science*. McGraw-Hill, New York, 1975.

[Vas98] Panos Vassiliadis. Modeling Multidimensional Databases, Cube and Cube Operations. In *Proc. of the 10th SSDBM Conf.*, Capri, Italy, July 1998.

[vDK99] Arie van Deursen and Tobias Kuipers. Identifying Objects using Cluster and Concept Analysis. In *Proc. of the Int'l Conference on Software Engineering*, pages 246–255, Los Angeles, CA, 16–22 May 1999.

[Wes96] Douglas B. West. *Introduction to Graph Theory*. Prentice-Hall, 1996.

[Wig97] T. A. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *Proc. of the 4th Working Conference on Reverse Engineering*, pages 24–32, Amsterdam, Netherlands, 6–8 October 1997.

[WTMS94] Kenny Wong, Scott R. Tilley, Hausi A. Müller, and Margaret-Anne D. Storey. Structural Redocumentation: A Case Study. *IEEE Software*, 12(1): 46–54, January 1994.

[YHC97] Alexander S. Yeh, David R. Harris, and Melissa P. Chase. Manipulating Recovered Software Architecture Views. In *Proc. of the 19th Int'l Conf. on Software Engineering*, pages 184–194, Boston, Massachusetts, USA, May 1997. Springer.