Louisiana State University
LSU Digital Commons

LSU Historical Dissertations and Theses | Graduate School

1996

A Reusability Model That Creates Design Frameworks Using a Formal Specification Clustering Approach.

Youwen Ouyang
Louisiana State University and Agricultural & Mechanical College

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

Recommended Citation

Ouyang, Youwen, "A Reusability Model That Creates Design Frameworks Using a Formal Specification Clustering Approach." (1996). *LSU Historical Dissertations and Theses*. 6361.
https://digitalcommons.lsu.edu/gradschool_disstheses/6361

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

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

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

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

A REUSABILITY MODEL THAT CREATES DESIGN FRAMEWORKS USING A FORMAL SPECIFICATION CLUSTERING APPROACH

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
Youwen Ouyang
B.S., Peking(Beijing) University, P.R.China, 1986
M.S., Xiamen University, P.R.China, 1989
M. Ap. Stat., Louisiana State University, 1994
December 1996

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

UMI Number: 9720375

UMI Microform 9720375
Copyright 1997, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized copying under Title 17, United States Code.

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

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

To my parents, my husband, and son

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

Acknowledgments

First of all, I thank God for His unconditional love and wonderful plan for my life. When I look back to this past six years, it is amazing how God has helped me overcome difficulties in many situations. He has also demonstrated His love through the loves of many new friends, especially those from The Chapel on the Campus.

I would like to extend my sincere appreciation to Dr. Doris L. Carver who has not only taught me fundamental knowledge in software engineering, but also guided me in searching for research direction and pursuing my goal. Her dedication and commitment to work have inspired me and evoked my deepest respect and admiration. She always knows exactly when to cool my head and when to encourage me. I am very fortunate to have her as my advisor, both scientifically and otherwise. I would also like to thank her for allowing me to use the facilities and equipment of the Software Engineering Laboratory.

I also express my deep appreciation to Dr. Barry Moser who has helped me in selecting appropriate courses in statistics to build up a sound foundation. In particular, his class in "Multivariate Analysis" broadened my perspectives of statistical application in computer science. Appreciation is also extended to all members of my doctoral advisory committee, Drs. Iyengar, Jones, Kraft and Lax, for taking the time to review my dissertation and offer their invaluable suggestions.

I owe special thanks to Dr. Iyengar for supporting me in attending a professional conference to present a paper and interact with other computer scientists. I would like also to thank Dr. Tyler and the assistantship committee for extending my

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

assistantship so that I could afford to stay in this Ph.D. program. To Ms. Kathy Traxler I owe special thanks for her consideration in my assistantship duty assignment. To Ms. Laura Courter, my excellent supervisor in teaching and dear friend in many other ways, I owe very special thanks for her tolerance and cooperation. Her dedication to quality of work has made it so much easier for me to accomplish my duty assigned.

The Avants and the Hays have opened their homes and hearts to adopt my family as part of theirs. Their concern and love have helped me adjust to Baton Rouge. I owe very special thanks to Mrs. Katie Avant, who has always taken the time to listen to my problems and share my happiness. I am grateful that God has arranged my meeting with this very special elder sister that I have always wanted.

The generous and enveloping love of my dear husband, Qun Zhu, has made my life and study as a Ph.D. student a great experience. He has shouldered my responsibilities, thereby freeing me to devote my time to research. He is also very understanding and tolerant. His consistent and unconditional love have been my greatest source of solace, strength and security. I am proud of him for his endurance for six years of my studies. I would like to also thank my beloved son, Bryan Zhu, who has always been an angel who removes shadows in me.

Finally and very importantly, I thank my parents who have taught me to be honest, to convert my weakness into strengths, and to always do my best. I owe deep appreciation for all their sacrifice especially when they decided to have my mother come to the U.S. to help me take care of my new born son. I would not have been able to do this Ph.D. without their moral support and guidance.

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

Table of Contents

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

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

List of Tables

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

List of Figures

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

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

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

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

Abstract

Software reuse has been advocated as a technique with great potential to increase software development productivity, reduce development cycle time, and improve product quality. Challenges for successful reuse include populating the repository with the right type of components, representing and organizing the components in a way that the components are easy to be retrieved, and providing mechanisms to compare the candidate components with the requirement of the new component and to assist adaptation. While many existing researches are emphasizing one or two challenges, this research proposes a reusability model that targets all challenges in reuse. The inspiration for the model is group technology which identifies and exploits the similarities in the parts to be manufactured and the sequence of machines that are necessary for the processing of those products. The Requirement Reusability Model (RRM) is proposed in this research to capture the aspect of reuse that a component can be constructed by modifying another component.

There are two major phases in RRM. At the Reusability Analysis (RA) phase, a measurement space is defined to represent the functional semantics of the components based on the formal specifications of the components. Clustering analysis is employed to group the components that are similar in function into the same clusters. At the Reusability Realization (RR) phase, the generic requirements for clusters are automatically extracted to create reusable frameworks for the clusters. The frameworks created are useful for constructing the implementation of individual transactions in the

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

same cluster. Guidelines for adaptation from the frameworks to actual implementation are also provided.

An automated system, called REST (a Reusers' assiSTant) implements RRM. The input to REST is the formal specification of a relational database system. Based on the input, REST produces a repository of reusable frameworks. The products of REST also include a data dictionary and a transaction dictionary. Each entry of the transaction dictionary indicates the name of a transaction, the description of the functional semantics of the transaction, the cluster to which the transaction belongs, and a suggested framework to be reused for the implementation of the transaction.

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

Chapter 1

Introduction

1.1 The Problem

Software techniques have improved significantly over the past 30 to 40 years, from primitive machine languages to sophisticated programming languages and tools for system configuration. Simultaneously, the size of software and the complexity of applications have grown explosively. However, software productivity and quality have not kept up with the demand that customers have placed on the software industry. Common complaints about software systems include late delivery, unreliability, and unfulfilled requirements. In addition, large quantities of existing software must be maintained. Due in part to a lack of systematic documentation, the cost of maintenance often exceeds the original cost of development. Software engineers face the challenge of developing high quality, easy to maintain systems and, at the same time, developing systems as quickly and cheaply as possible.

One way to improve the quality of software systems is to include formal methods to describe system properties in the development. Formal methods, which can be used to reveal ambiguity, incompleteness, and inconsistency in systems, assist in the production of high quality products [Win 90]. However, there has been a reluctance to

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

accept formal methods in industrial practice due to the fact that formal methods are based on mathematical theories. A common problem regarding the application of formal methods to real problems is that many people find writing formal specifications difficult. Such a problem implies that writing a formal specification is more expensive than an informal one. Success in introducing formal methods into industry depends on making those methods cost-effective in the overall context of industrial software engineering practice [Gar 90].

As a different approach to the software problems, software reuse has attracted great interest in recent years. Several decades of intensive research in software engineering and artificial intelligence have left few alternatives but software reuse as the (only) realistic approach to cause the gains of productivity and quality that the software industry needs [Mil 95]. Instead of searching for ways of writing code faster, reuse engineers are looking for ways of writing less code [Boe 87]. Therefore, reusable software has the potential to increase software productivity by an order of magnitude or more [Hor 89].

Current reuse methodologies require a repository of artifacts that can provide a sufficient number of components, over a spectrum of domains, that can be reused as is or easily adapted [Maa 91]. Figure 1.1 summarizes the current reuse process. The repository receives components from existing systems and domain analysis. The designer requests components from the repository to build a new system. The cost of reuse comes from maintaining a repository of reusable parts, retrieving parts from the repository, and adapting parts to fit the new requirement. The first challenge in reuse is

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

to populate the repository with the right type of components. Besides code segments, other artifacts that are produced in the earlier phases of the software lifecycle, such as requirement specification and design artifacts, can also be reused to produce even more gains [Tra 90, Pri 93, May 95]. When designing a new system, designers need to be able to find a component faster and easier than to write it from scratch. The second challenge of software reuse comes from representing the components and organizing them in a way that the components are easy to be retrieved. Once a component is found, a comparison needs to be performed to see whether it can be used directly. The third challenge of software reuse is to provide mechanisms to compare the candidate components with the requirement of the new component. In addition, guidelines need to be provided to assist any adaptation necessary. This research targets these major challenges of software reuse, that is building and maintaining a repository of reusable components and providing guidelines to support adaptation.

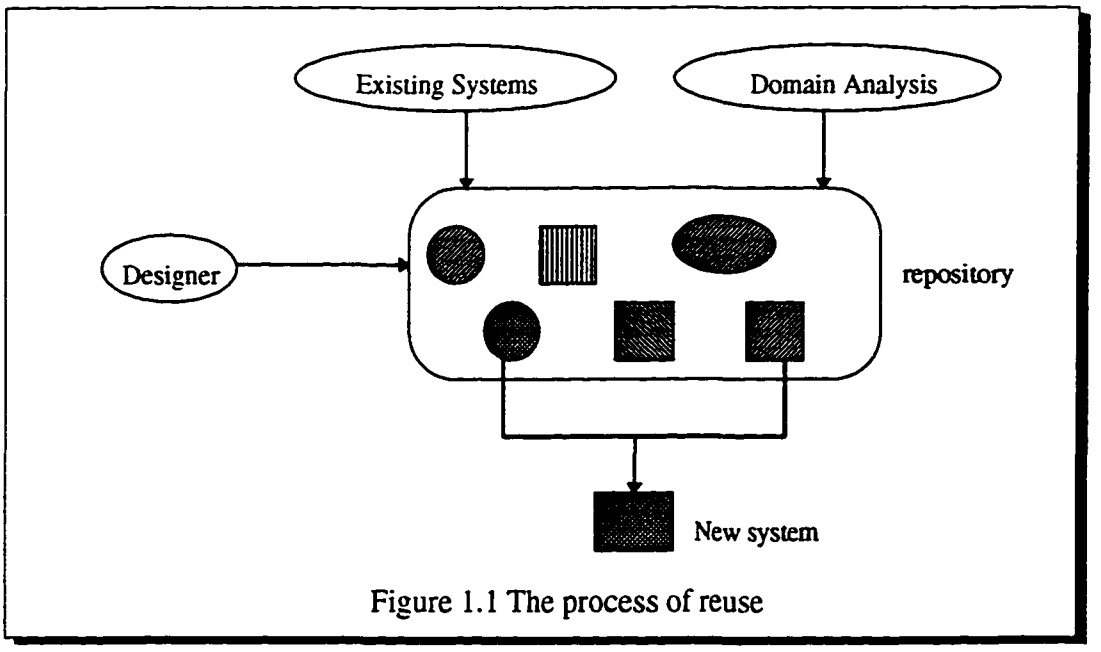

Figure 1.1 The process of reuse

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

In addition, most research in software reuse today deals with cross-system reuse, that is designing and implementing a new system by reusing components found in a repository populated by components either salvaged from existing software system or specially designed as the result of domain analysis. Little attention has been paid to reusability within a single system, except for class inheritance and function calls.

1.2 A Solution

This research presents a complete reusability model for relational database applications. The basic elements of interest are transactions which, according to [Dat 90], are *logical units of work* upon the contents of a database system. The relational database domain is chosen because of its wide use in industry. Examples of the application include accounting systems and membership administration systems for associations. In addition, formal methods are used in practice to specify the requirements of relational database systems. On the other hand, there is a lack of effort from the reuse community for business applications [Jon 95].

A Requirements Reusability Model (RRM) is defined in this research. RRM is intended to capture the aspect of reuse that a component can be constructed by modifying another component [Ost 92]. RRM reduces the cost of developing a software system through reusing artifacts that are available within the system. There is no assumption of an existing repository at the time of software development. The only assumption is that a formal specification is generated as the results of the requirement specification phase and written according to "plans" and "discourse rules". The formal

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

specification works as a contract between software engineer and client. The reusability analysis is performed on the formal specification of the system. A repository of reusable artifacts is available at the end of the reusability analysis. RRM recognizes the potential of reusability within a single system at the end of requirement specification stage and facilitates reuse activity at the implementation stage.

There are two major phases in RRM: the Reusability Analysis (RA) phase and the Reusability Realization (RR) phase. At the RA phase, a measurement space is defined to describe the functional semantics of the transactions. Each transaction is measured by a *descriptor set* which is composed of <descriptor, value> pairs. The descriptor set is extracted systematically from the formal specification of a transaction. Clustering analysis is applied to the descriptor sets of the transactions to assure that transactions that belong to the same cluster are more homogeneous in function than transactions that belong to different clusters. At the RR phase, the generic requirements for clusters are automatically extracted to generate reusable frameworks for the clusters. Since the transactions grouped into the same cluster are similar in function, the generic requirements extracted reflects common functionality of the transactions in the cluster. Therefore, the frameworks generated are useful for constructing the implementation of individual transactions in the same cluster. Guidelines for adaptation are also provided so that the repository contains components that are usable for the implementation of individual components. Figure 1.2 shows the main steps of RRM. Rectangles are used for intermediate documents and ovals are used for actions.

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

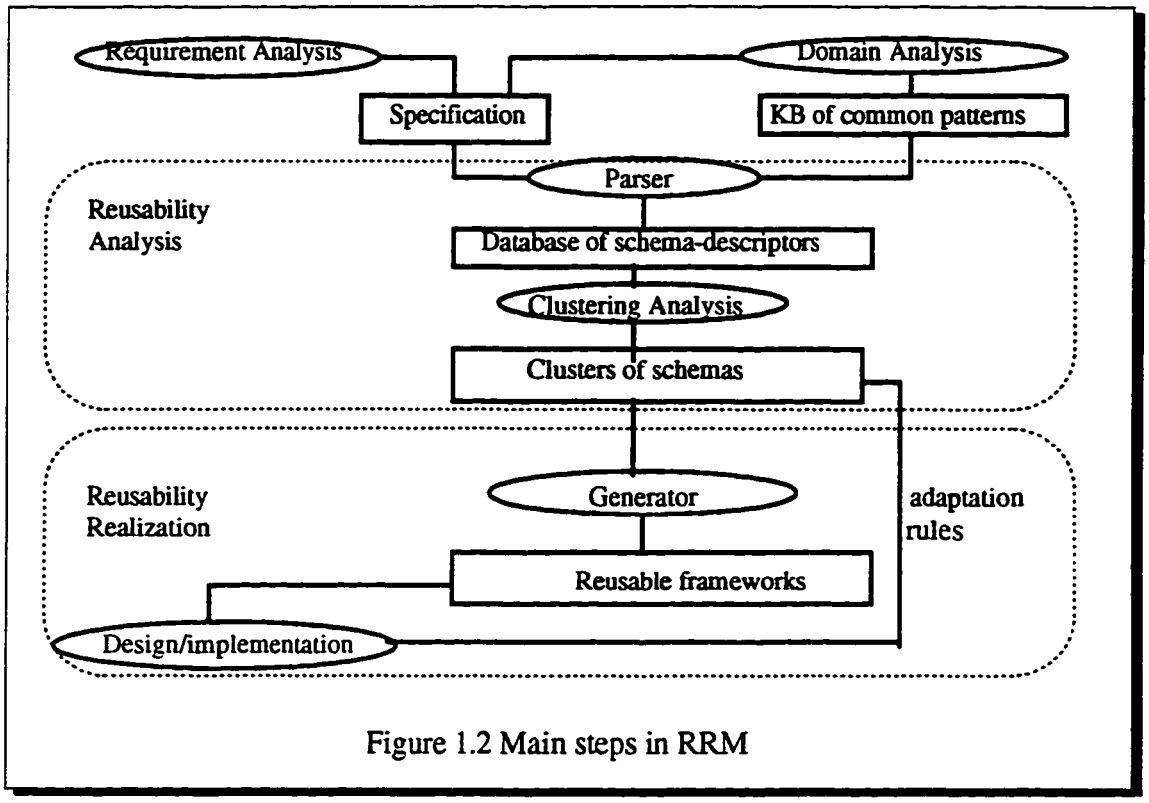

Figure 1.2 Main steps in RRM

An automated system, called REST (a REusers' assiSTant), implements RRM. For a relational database system, REST produces a repository of reusable frameworks. The products of REST also include a data dictionary and a transactions dictionary. The entries of the data dictionary are attributes. Information available for each attribute include the name of the attribute, the domain that the attribute draws values, the relation to which the attribute belongs, and constraints applicable to the attribute. Each entry of the transactions dictionary indicates the name of a transaction, the description of the functional semantics of the transaction, the cluster to which the transaction belongs, and a suggested framework to be reused for the implementation of the transaction.

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

1.3 Motivation

1.3.1 Why formal methods

Formal methods are mathematically based techniques for describing system properties to assure accuracy and therefore improve quality and reduce work. Anyone involved in any stage of system development can make use of formal methods [Win 90]. Formal methods can be used from the initial statement of a customer's requirements, through system design, implementation, testing, debugging maintenance, verification, and evaluation. A method is formal if it has a sound mathematical basis, typically given by a formal specification language. Its role in software development is important because using formal methods for specifying and verifying software can, in principal, offer a greater assurance of correctness by revealing ambiguity, incompleteness, and inconsistency in a system.

One tangible product of applying a formal method is a formal specification which serves as a contract, a valuable piece of documentation, and a means of communication among a client, a specifier, and a designer. [Win 90] Specifiers write, evaluate, analyze, and refine specifications. They prove that the specification preserves certain properties. Customers of the software system read the specification to assure that the properties meet their requirements. Designers take the specification which is more formal than most customers' statements, and try to develop a software system that matches the specification. Software products that are developed with the help of

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

formal methods are expected to be more consistent with the users needs than those without.

Despite the above advantages of using formal specifications, they are not widely accepted in software development practice. The main reason is the difficulty, which leads to high costs, of writing formal specifications due to their mathematical basis. As a result, management does not see a cost saving by using formal methods. This research is intended to amortize the cost of writing a formal specification.

1.3.2 Why reuse

Software reuse is viewed as a promising approach for addressing numerous software problems such as late deliveries, unreliable software, and high maintenance. However, despite almost two decades of intensive research, the routine production of software under acceptable conditions of quality and productivity remains an unfulfilled promise. Few reports have been made on benefits of software reuse in development of real life software systems. Current reuse requires facilities such as a defined application domain, a library of reusable software components and a system for instantiating the required components into an application system. The cost of building such a reuse source facility and stocking it with software is a capital investment for software reuse.

There exists extensive literature on representing components and retrieving them from a library. However, little work has been done to overcome the difficulty of creating the right components that are both useful and usable. Building such a system may even be more costly than writing new software [Gar 95], not to mention the cost of adapting components to satisfy the new requirement. This research is intended to

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

derive a systematic approach to build a repository of components that are both useful and usable for the development of a single system.

1.3.3 Why clustering

Another challenge of software reuse comes from organizing the collected components. Different classification approaches have been proposed to organizing software collections into a software library system to facilitate queries and retrieval [Dev 90, Jon 88]. Classification is a statistical technique concerned with separating distinctive sets of objects and allocating new objects to previously defined groups. The immediate goal is to sort objects into two or more labeled classes. The emphasis is on deriving a rule that can be used to optimally assign a new object to the respective labeled classes. The structure of the library is determined, usually by domain experts as in most facet classification schemes, before the components are collected and assigned to different categories. Clustering on the other hand is distinct from classification methods in that it makes no assumptions concerning the number of groups or the group structure. A substantial amount of effort in the development of clustering methods has been devoted by statisticians over decades to find "reasonable" clusters without having to look at all configurations. The basic objective is to discover a natural grouping of the items. The formal specification of an application specifies the functional semantics of individual transactions in the application. In this research, clustering analysis is applied to the basic elements of the formal specification so that the resulting clusters consist of transactions that are similar in function, while transactions from different clusters are dissimilar in function.

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

In addition, while a great deal of research of software reuse has emphasized the building of new software systems from existing software, little has been done on reducing the repetitive efforts within the development of a single software system with the except of function calls and class inheritance. If *A* and *B* represent two components with similar functionality which are to be developed in a new system, under current reuse schemes, two trips to the repository will be needed. One trip for finding a candidate component, *C*, in the repository to be reused by *A* and the other trip for finding *C* again to be reused by *B*. By grouping *A* and *B* before retrieval, only one trip to the repository is necessary. In addition, the less adaptation required from the reused component to the desired one, the greater the saving through reuse [Pri 93]. Components developed in the same system usually share many common features such as company policies, hardware, implementation language, and platform. Therefore, reusing components that are developed for the same system requires less adaptation than reusing components that were developed for other systems.

1.4 Background

1.4.1 The Z specification language

Wing [Win 90] classifies formal specification languages into two categories: model-oriented or property-oriented. A model-oriented specification language defines a system's behavior directly by constructing a model of the system in terms of mathematical structures, while a property-oriented specification language defines the system's behavior indirectly by stating a set of properties. Z is a formal specification

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

language that is based on first order logic and set theory. The language was developed by the Programming Research Group at Oxford University, UK which can be used in both model-oriented and property-oriented styles. It is an established language which has been under development for over a decade and is currently being standardized. It is one of the most well-known formal specification languages in Europe and has been adopted in many projects both in academia and in industry [Hal 90]. This section introduces representations that are necessary for the understanding of this research. The definitions come from [Wor 92]. Interested readers can refer to extensive literature sources, including a user manual [Spi 92] and an introductory textbook [Pot 91].

The set theory on which Z is based is a typed set theory, that is to say every value must be assigned a type. One way to introduce a type into a Z specification is as a *given set* by putting the name of the set in square brackets, e.g. [City]. A type can also be defined by set enumeration, which defines a set by listing all its members. There are two sets of numbers that are particularly useful: *Integers* $\mathbb{Z}$ (the set of positive or negative whole numbers including zero) and *Natural numbers* $\mathbb{N}$ (the set of whole numbers from zero upwards). When we refer to a more restricted set of numbers, *subrange* is used to define numbers lying in a certain range. The other powerful means of defining a set is called *set comprehension*, which defines a set by stating a property that distinguishes its members from other values of the same type. Its general form is $\{\mathbf{D} \mid \mathbf{P} \bullet \mathbf{E}\}$ where **D** denotes some declarations, **P** is a predicate constraining the values and **E** is an expression denoting a term. The set introduced consists of all values of the term **E** for everything delcared in **D** satisfying the constraint **P**. Sometimes the

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

term and the preceding heavy dot are omitted. In such a case the term is taken to be equivalent to **D**. The constraint and the preceding vertical bar are often omitted, in which case, no further constraint is applied to the term.

To assign a particular type to a *variable*, the name of the variable is followed by a colon and then by the name of the set from which the variable takes its values. Two sets of values of the same type are equal if and only if the two sets have the same members. One set is said to be a *subset* of another if all the members of the one are also members of the other. The *union* of two sets contains all members of both sets and the *intersection* of two sets contains common members of the two sets. The *power set* of a set A, denoted by $\mathbb{P}$ A, is the set of all its subsets.

The basic elements in Z specifications are schemas. Each schema describes an abstraction of the application domain or an operation on an abstraction. Typically, a Z document contains data schemas to specify the data structure of the system and operation schemas to describe state transitions. Each schema contains two parts [Wor 92]: the declaration part declares variables and the predicate part expresses requirements about the values of the variables. In this thesis, schemas are written in a vertical format as illustrated by the template in Figure 1.3.

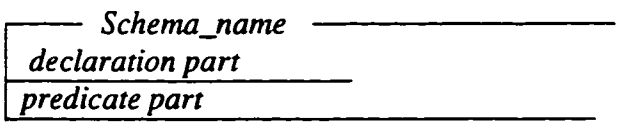

Figure 1.3 Template of a Z schema

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

Schemas easily get cluttered with detailed information. Z provides conventions that can reduce the amount of information and draw attention to the main points of the behavior being described. Standard conventions used in the thesis are schema decoration and schema inclusion which include the Delta (Δ) and Xi (Ξ) conventions. The definitions of each of these conventions follow:

Schema decoration provides a systematic way to represent the states before and after an operation. Undashed names are used to denote the values of the components of the state before the operation. Dashed names are used to denote the values of the same components of the state after the operation. Names with a '?' at the end are used to denote the values of inputs to the operation. Names with a '!' at the end are used to denote the values of outputs from the operation.

Schema inclusion refers to the situation that the name of a schema is included in the declaration part of another schema. By including the name of a schema, the declarations and constraints applicable to the variables of the schema are also included in the new schema. There are two inclusion conventions in Z:

- **The Delta (Δ) convention** -- The inclusion of the dashed and undashed version of a schema S can be abbreviated to the single name ΔS which indicates a change in the value of one or more attributes of the schema after an operation has been completed.
- **The Xi (Ξ) convention** -- Another way of abbreviating the inclusion of dashed and undashed version of schema S is ΞS which implies that there is no change in the value of any attribute of the schema after an operation has been completed.

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

1.4.2 Relational database systems

A majority of the database systems developed over the past few years are relational. The basic concepts and structure of a relational database are introduced in this section. Interested readers can refer to [Dat 90] for more comprehensive information.

A *database system* is basically a computerized record-keeping system which maintains information and makes the information available on demand. It can be regarded as a repository for a collection of computerized data files. The user of the system is given facilities to perform a variety of actions on such files, including :

- Adding new files to the database;
- Inserting new data into existing files;
- Retrieving data from existing files;
- Updating data in existing files;
- Deleting data from existing files;
- Removing existing files from the database

Four major categories of database systems are *inverted list, hierarchic, network* and *relational.* They are categorized according to the data structures and operators they present to the user. An inverted list database contains a collection of files that are divided into records and fields. The records of an inverted list file are ordered in some physical sequence. For a given file, any number of search keys can be defined. Such keys permit both direct and sequential access to records in the file. A hierarchic database consists of an ordered set of multiple occurences of a single type of tree. A

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

record type is made up of field types. A tree type consists of a single "root" record type, together with an ordered set of zero or more dependent subtree types. The entire tree type thus consists of a hierarchic arrangement of record types. The herarchic sequence defines a total ordering for the set of all records in the database. The network database can be regarded as an extended form of the hierarchic database. The principal distinction between the two is that a child record has exactly one parent record in a hierarchic structure but can have any number of parents (possibly zero) in a network structure. A relational system is a system in which the data is perceived by their user as relations. The operators at the user's disposal are operators that generate new relations from old relations.

The design of a relational database application, like other database applications, includes data definition and data manipulation. Data definition specifies the structure of the system, which defines the following aspects of the database:

- A *domain* is a set of individual data values which are of the same type. Domains are pools of values from which the actual values appearing in attributes are drawn. At any given time, there will typically be values included in a given domain that do not currently appear in any of the attributes that correspond to that domain.
- A *relation* on domains $D_1, D_2, \ldots, D_n$ (not necessary distinct) consists of a heading and a body. The *heading* consists of a fixed set of *attribute names* $A_1, A_2, \ldots, A_n$ such that attribute A_i corresponds to exactly one of the underlying domains D_i (i = 1, 2, ..., n). The *body* consists of a time-varying set of *tuples*, where each tuple in turn consists of a set of attribute-value pairs (A_i, v_i), one pair for each attribute A_i

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

in the heading. V_i is a value from the unique domain D_i that is associated with the attribute A_i.

- Let R be a relation with attributes $A_1, A_2, \ldots, A_n$. The set of attributes K = ($A_i, A_j, \ldots, A_k$) of R is said to be a *primary key* of R if it satisfies the uniqueness property, i.e., at any given time, no two distinct tuples of R have the same value for K and none of the attributes in K can be discarded without destroying the uniqueness property. The primary keys provide the sole tuple-level addressing mechanism within the relational model. That is, the only system-guaranteed way of pinpointing some individual tuple is via the values of the primary keys.
- A *foreign key* is an attribute (or attribute combination) in a relation R_2 whose values are required to match those of the primary key of some relation R_1. Foreign-to-primary-key matches represent *references* from one relation to another. They are the "glue" that hold the database together.

Data manipulation allows users to communicate with information stored in the system. As stated earlier, a relational system is a system in which the data is perceived by the user as relations. The user of the system is given facilities to perform a variety of operations on relations, including :

- Inserting new tuples into existing relations;
- Retrieving data from existing relations;
- Updating attribute values in existing relations;
- Deleting tuples from existing relations;

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

At any given state of the application, two integrity rules of the relational model must be always satisfied. The first rule is the entity integrity rule which restricts any attribute participating in the primary key of a relation from being null. The other rule is the referential integrity rule which states that every value of a foreign key must either be null or equal to the value of its corresponding primary key value in some tuple.

1.4.3 Case study problem

The case study problem used in this dissertation is a relational database system which is an extended version of the example presented in [Bar 94]. In this example, employees are hired by departments to work on one or more projects controlled by a given department for a certain number of hours. Additional constraints include: employees should be paid at least minimum wage, all employees must work at least four hours on each project they work on, and a project should end no earlier than it starts. We extend the example by adding attributes to relations. The case study involves typical database transactions which cover all possible basic database operations. The case study is sufficient enough to illustrate the method and provide evidence of benefit achieved by RRM.

The data structure of the example database is presented in this section. The conceptual model of the database is presented in Figure 1.4 using an Entity Relationship (ER) model [Che 70]. Rectangular boxes are used to denote the entity types in question. Diamond-shaped boxes are used to denote the relationships. Ovals are used to indicate attributes, among which primary keys are underlined. Lines that

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

connect relationships and entities are labeled by numbers to indicate whether the relationship is one-to-one, one-to-many, or many-to-many.

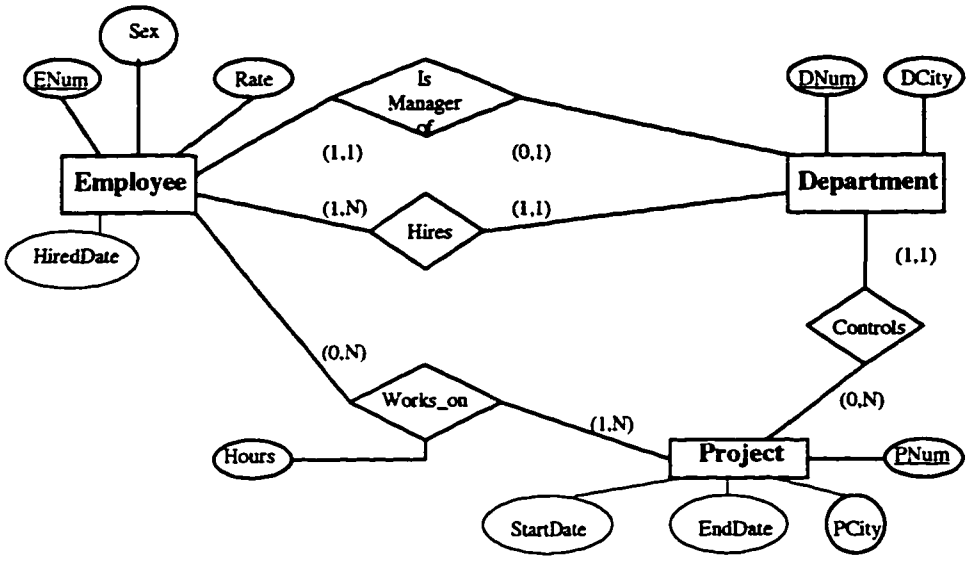

Figure 1.4 ER model of the example database

According to [Che 71], entities are transferred to relations and additional relations are introduced for many-to-many relationships. In this section, data definitions of the example database are specified in Z. First, Figure 1.5 defines the domains from which attributes draw values.

Given types: [CITY] [DATE]

ENUM == N DNUM == N PNUM == N

SEX ::= Male | Female RATE == REAL HOURS == REAL

Figure 1.5 Domains for attributes

Then the relations are specified. In the example database, there are four relations: *Employee*, *Department*, *Project* and *Work*. Figure 1.6 defines the headings of relations which include the name of attributes and the underlined domains from which the attributes get their values. Figure 1.7 lists schemas for the relations. The declaration

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

part of each schema indicates the tuple type of a relation. The predicate part of each schema defines intra relation constraints that are applicable to values of attributes.

EMPL = = [ENum : ENUM; Sex : SEX; Rate : RATE; HiredBy : DNUM]
DEPT = = [DNum : DNUM; ManENum : ENUM; DCity : CITY]
PROJ = = [PNum : PNUM; CtrlDate : DNUM; StartDate : DATE;
EndDate : DATE; PCity : CITY]
WORK = = [ENum : ENUM; PNum : PNUM; Hours : HOURS]

Figure 1.6 Tuples in the database

Employee

empls : P EMPL

$\forall e : empls \bullet e.ENum \neq Null \wedge$
$\forall e_1, e_2 : empls \bullet e_1.ENum = e_2.ENum \Leftrightarrow e_1 = e_2 \wedge$
$\forall e : empls \bullet e.Rate \geq 4.5$

Department

depts : P DEPT

$\forall d : depts \bullet d.DNum \neq Null$
$\forall d_1, d_2 : depts \bullet d_1.DNum = d_2.DNum \Leftrightarrow d_1 = d_2$

Project

projs : P PROJ

$\forall p : projs \bullet p.PNum \neq Null \wedge$
$\forall p_1, p_2 : projs \bullet p_1.PNum = p_2.PNum \Leftrightarrow p_1 = p_2 \wedge$
$\forall p : projs \bullet p.EndDate \geq p.StartDate$

Work

works : P WORK

$\forall w : works \bullet w.ENum \neq Null \wedge$
$\forall w : works \bullet w.PNum \neq Null \wedge$
$\forall w_1, w_2 : works \bullet (w_1.PNum = w_2.PNum \wedge w_1.ENum = w_2.ENum) \Leftrightarrow w_1 = w_2 \wedge$
$\forall w : works \bullet w.Hours \geq 4$

Figure 1.7 Data schemas of relations

Operations applied to the database are also specified using Z schemas. Since the operation may affect the contents in more than one relation, and all constraints need to

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

be satisfied at any given state of the application, we specify the database by a single schema which includes all relations along with inter-relations constraints including built-in constraints applicable to foreign keys. The schema that specifies the database is displayed in Figure 1.8.

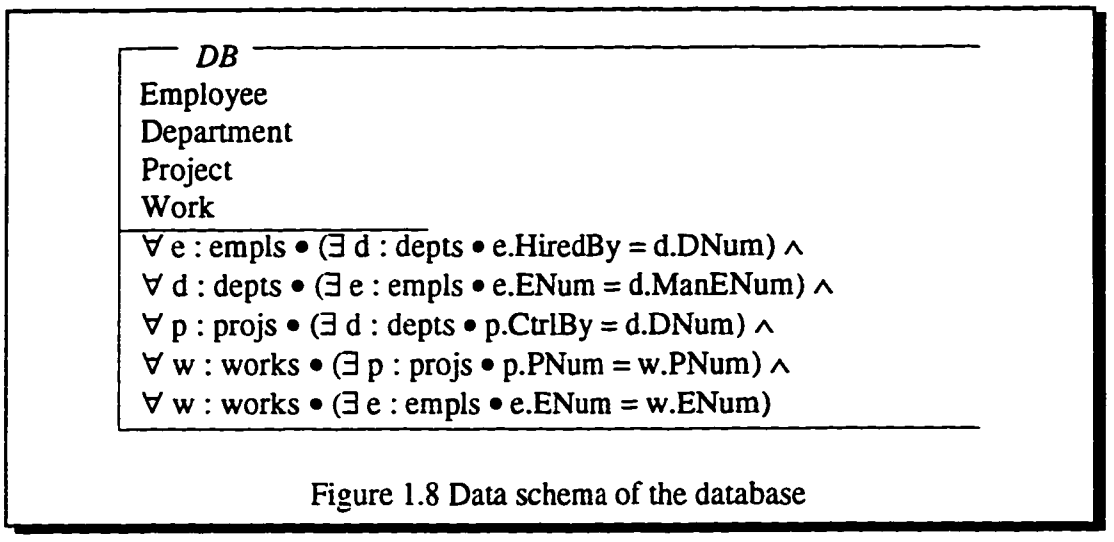

Figure 1.8 Data schema of the database

Additional schemas of transactions applied to this database are given in Chapters 4 and 5 to illustrate the two phases (RA and RR) of RRM. A complete treatment of the case study and the results are presented in Chapter 6.

1.5 Overall Organization

Chapter 1 introduced the motivation for conducting this research and the preliminaries needed to understand the work. In particular, the basic concepts of the Z specification language, relational database systems are presented. In addition, the case study problem is defined and explained in the first chapter. An overview of the major steps in RRM is also presented.

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

Chapter 2 serves as a literature survey of the existing techniques used in organizing software components. It first summarizes the current reuse methodologies and then compares major component retrieval methods.

Chapter 3 presents patterns used to specify important basic operations of relational database applications in order to achieve rigorous specification in a formal language. A measurement space is then defined so that the functional semantics of each transaction can be described using a descriptor set. Operations on the descriptor sets are defined followed by a similarity measurement algorithm. The single linkage method is employed to analyze similarity indices and to generate cluster hierarchies.

Chapter 4 presents the techniques for the automatic extraction of generic requirements from descriptor sets of transactions in the same cluster. Algorithms are provided to determine the minimum, maximum and majority requirements for each cluster. Frameworks are generated to accomplish the generic requirements. The implementation of a transaction selects one of the three frameworks as the best candidate and adapts the selected framework to meet the requirements of the transaction. Finally, the guidelines for implementing a new requested transaction through reusing the frameworks are provided.

In Chapter 5 a case study is presented to illustrate the process of RRM. Example transactions are specified using the formal specification language Z and the corresponding descriptor sets of Z schemas are generated. Similarity indices among the Z schemas are presented along with the result of the clustering analysis. The process

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

continues by generating reusable frameworks for each cluster and ends with examples showing the adaptation procedures for implementing individual Z schemas.

Finally, Chapter 6 presents the conclusions reached from the research. It summarizes the research and devotes special attention to the benefits of the model. The chapter closes with extensions and future work.

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

Chapter 2

Related Work

The idea and practice of software reuse have been around since humans began to solve problems. Code, subroutines and algorithms have been reused since programming was invented. But they have been used informally. Only when reuse can be conducted systematically and formally will a substantial quality and productivity payoff be achieved. It was Doug McIlroy who first introduced the concept of systematic software reuse. He proposed an industry of off-the-shelf standard source-code components and envisioned the construction of complex systems from small building blocks available through catalogs [McI 68]. Software reuse has since been advocated as a technique with great potential to increase software development productivity, reduce development cycle time, and improve product quality [Agr 87, Bro 87, Boe 88]. Instead of searching for ways of writing code faster, software reuse looks for ways of writing less of it: we could reuse the processes and products of previous development efforts in order to develop new applications [Boe 87]. There are promising reports suggesting high potential reuse rates. In the early eighties, Langergan and Grasso estimated that 60% of business applications can be standardized and reused. Jones' analysis on some applications shows that they contain less than 15

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

percent of application-specific-code while the remaining 85 percent comes from common, redundant and potentially reusable software components [Jon 84].

An extensive amount of research has been conducted in the area of software reuse. In this chapter current methodologies for software reuse are presented. Existing organization schemes and retrieval methods used in libraries of software components are then reviewed

2.1 Reuse Methodologies

In general, an adequate library of candidate components is crucial to the success of software reuse. By adequate, the library should have the following characteristics [Maa 91]:

1) provide a sufficient number of components, over a spectrum of domains, that can be reused as is or easily adapted,
2) be organized such that the existing code closest to the users' needs is easy to locate.

In this section, a survey of research on the first characteristic is presented. As illustrated in Figure 2.1, current reuse methodologies are discussed from three perspectives: how to populate a repository, what is the granularity of the components in the repository, and what is the level of abstractions of the components in the repository. In Section 2.1.1, we present the methods used to populate the repository. The granularity and the levels of abstraction of the components in the repository are discussed in Section 2.1.2 and Section 2.1.3, respectively.

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

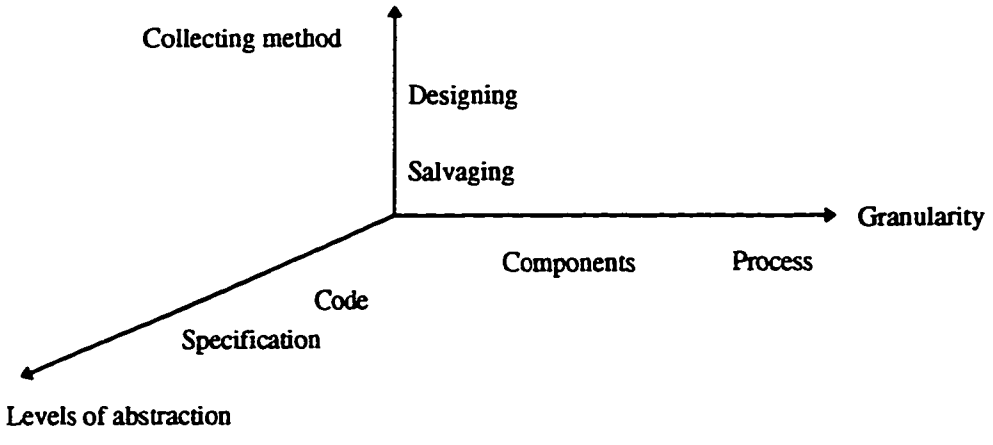

Figure 2.1 Three points of view on reusable library

2.1.1 Design-for-reuse vs. design-by-reuse

The debate between design-for-reuse and design-by-reuse is about how to solve the problem of populating the repository of reusable artifacts. According to a design-for-reuse perspective, a repository must be populated with reusable components that are properly defined to be exploitable for designing new applications. Reuse libraries should not be populated with randomly harvested components. Researchers favoring this perspective claim that the quality required for reusable artifacts exceeds the quality of custom-developed components, and few of the existing components will qualify to be included in a base of reusable assets or will be worth expending effort. Existing components are typically not interoperable (because they are based on conflicting assumptions), not composable (because they have ad-hoc interfaces), and hard to be transformed [Gar 95].

Castano and De Antonellis propose a methodology for populating a repository of reusable components by *application independent* specifications [Cas 93]. The

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

reusable specifications are defined by abstracting common properties and behavior from different existing conceptual specifications. A collection of available conceptual schemas belonging to one or more applications in one or several domains is classified and arranged into categories. The classification model is derived based on indexing criteria and clustering techniques adapted from the information retrieval area. The reusable resources, also called *generic resource classes*, are extracted by analyzing the conceptual schemas in each category based on pair-wise comparisons between the identified resource classes and on the computation of the level of semantic affinity between the conceptual schemas. Then a threshold value is defined for a given cluster and only the class pairs whose affinity exceeds the threshold are grouped into an *affinity set*. The resulting affinity sets are proposed to the application engineer who analyzes the meaning and the role of each object included in the affinity set. The corresponding generic classes for affinity sets are defined by factoring out the properties common to all the objects belonging to the sets, taking into account the existence of synonyms.

In contrast to the design-for-reuse approach, according to a design-by-reuse perspective, the repository can be populated by tailoring existing components extracted from existing software systems. The argument here is that the idea that reusable components need to be designed rather than discovered is economically dissatisfying to companies which have access to a large inventory of existing software components. Researchers attempt to identify reusable components from existing software systems

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

based on a number of heuristics such as frequency of reference, level of coupling and cohesion.

SRSGEN (Software Reuse System GENerator) is a set of front-end tools for automatically populating a single database system, the Software Reuse System (SRS), from a set of unorganized software components [Ku 93]. SRSGEN is written primarily in the AWK language, a programming utility of the Unix Operating System and runs on a SUN SPARC workstation. It is a powerful utility for extracting free-form text from source code and other types of ASCII files. A user enters a table of attributes and its initial constraints in a SRSGEN specification (SPEC) file, pertaining to the unique requirements of the projects. A custom-made SRSGEN tool set is built within seconds. The toolset then searches for blocks of free-form text that meet the criteria. Once a comment block is identified, a search for predefined fields of information is done within the comment block. A Unix shell script enables these tools to be run as a single batch processing function that recursively scans any number of directory locations. The net results include a well-structured data file that is ready to be loaded into the SRS database component of the system, a detailed data analysis, and a set of e-mail messages reporting the status of document registration to designated or default file owners.

Esteva and Reynolds describe the use of inductive learning techniques based on software metrics to identify reusable modules [Est 91]. The work is extended by Esteva to automatically identify reusable components from code [Est 95]. They codify design knowledge used by experienced software developers to access the reusability of a

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

software component. The conjecture is that components that are easier to comprehend will tend to be reused more often. The complexity of a component is measured by counting tokens of two types: components and objects. A name corresponds to a conceptual object manipulated by the program. Each occurrence of the name is a reference to the object. Objects that are referenced frequently are more likely to influence a program's results. The same argument can be made for the components in a program. On the other hand, the complexity of a component is based on the number of interactions between the component and its environment, and, to a lesser degree, on the internal complexity of that component. Components that are loosely bound tend to be easier to remove and to use in other contexts than those that depend heavily on other components or non-local data.

2.1.2 Granularity of reusable assets

Research in software reuse also emphasizes the granularity of the artifacts, which ranges from a statement to a process. A process is a set of partially ordered steps intended to reach a goal [Fei 92]. The larger the granularity, the larger the "win" for productivity [Tra 90].

AIRS is a system originally designed to reuse Ada packages but has evolved into a general tool for reuse [Ost 92]. Ostertag et al. propose reusing artifacts of two granularity: components and packages, where a package is a logical unit that groups a set of components that are tightly coupled. Each component in the repository is described by a set of features which are used to characterize the different aspects of the component. A feature is defined as a finite set of related values called terms. The

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

subsumption relation is intended to capture the idea that certain components can be built by composing several other components. A weighted directed acyclic graph is composed of components as nodes and arcs indicating that the source node subsumes the destination node with weights representing the estimated effort required to obtain the subsumed given the subsumer. And the candidate components are those have the shortest distance from the query component. The closeness relation is intended to capture the idea that a component can be constructed by modifying another component. As for the subsumption relation, a weighted directed graph is used to represent expected effort required to obtain the target terms given the candidate term. The closeness distance is measured by total distance for all features. The candidate package is chosen in a similar way.

Zaremski [Zar 95b] also considers two kinds of software components: functions (e.g. C routines, Ada procedures, ML functions) and modules where a module is a collection of functions that provide a set of operations on an abstract data type (e.g. C++ classes, Ada packages, ML modules). Functions and modules are represented in the similar way by two different kinds of semantic descriptions: signatures and specifications. Both exact match and various notions of relaxed match are defined to assist retrieval of functions and modules.

Hollenbach and Frakes [Hol 95] define process reuse as the usage of one process description in the creation of another process description. The 3Cs model [Lat 91, Tra 90] is adapted to represent reusable processes. The *concept* of a reusable process is an informal specification of the general information, the customer description

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

and the interface description. The *content* of a reusable process includes the procedural description utilizing both textual and graphical representations. The reusable process *context* describes the environment in which the process can execute. A pragmatical and systematical method is proposed to create a standard reusable process description, instantiate it for use on a project, refine its usage, and improve the process description for subsequent use based on feedback. A graphical notation is introduced to describe the process, its customers and an indicator which measures compliance in meeting customer requirements. The graph is divided vertically into columns where each column represents a participant in the process. The graph can also be divided into horizontal bands where each band represents a step of the process.

2.1.3 Levels of abstraction

Although different researchers propose different categorizations of reusable knowledge, it is generally agreed that reuse applies not only to source-code fragments, but also to all the intermediate work products generated during software development as illustrated in Figure 2.2. At the most basic (and successful) level, there are subroutine libraries, such as those for mathematical functions for C and FORTRAN. At a higher level, one can attempt to reuse designs or specifications and to reuse the knowledge behind design or implementation decisions [Lan 94]. Integration of software reuse into each stage of the software development life cycle can provide long-term progress in solving productivity and quality problems and, therefore, significantly reduce the cost of developing software. Krueger proposed a multilevel categorization of reusable information based on levels of abstraction, where reusable items of level i

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

are abstractions of reusable items of level i-1. And reuse occurs within the same level of the "abstraction hierarchy" [Kru 92]. Prieto-Diaz divided software reuse into idea reuse, artifacts reuse and procedures reuse according to the essence of the items to be reused [Pri 93].

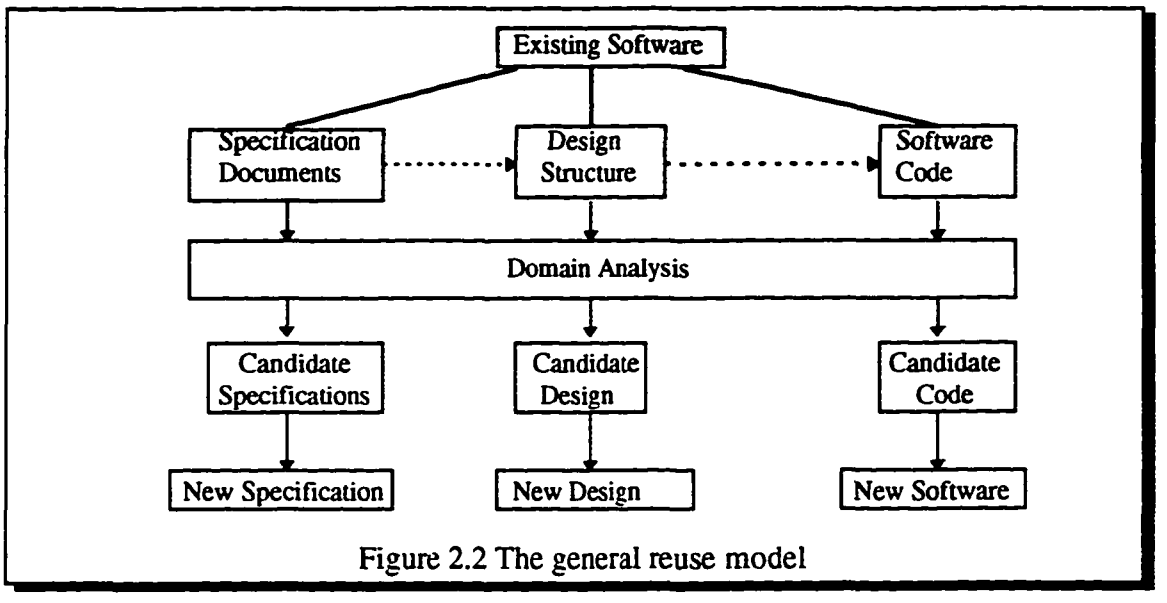

Figure 2.2 The general reuse model

As pointed out in section 2.1.1, Castano and De Antonellis populate a library of reusable components by application independent specifications from a design-for-reuse perspective [Cas 93]. The basic elements in the repository are generic resource classes and generic process classes. A "meta-of" link connects a meta-resource class with its corresponding generic resource class and a meta-process class with its corresponding generic process class. Meta classes describe how the generic classes can be reused and tailored. Suggestions are given in the form of properties in the meta class. The properties incorporate information concerning the most common design choices, according to the design history information of the clustered conceptual models.

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

2.2 Component Retrieval Methods

In section 2.1, a survey was presented on different ways of providing a sufficient number of components as an adequate reusable library. In this section, discussion is given on different ways of finding artifacts in the repository. As in a library of books, we can search components by names if we know them or even use wild card characters if we can guess them. However, since people use a variety of terms to refer to the same thing and different people may use the same term to refer to different things, organizing components merely by name can hardly provide satisfactory response to users' queries. Since we are reusing software components to achieve desired functionality, reuse engineers have tried to organize the libraries of software components based on semantics of components. Currently there are mainly four ways to search components based on semantics:

- Search by function prototype,
- Facet Classification,
- Search by profiles,
- Match through verification

Techniques using each of these four methods are presented in Sections 2.2.1, 2.2.2, 2.2.3, and 2.2.4, respectively.

2.2.1 Search by function prototype

Instead of the names of components, the search by function prototype method looks at the "signatures" of the components in the library and compares them with the signature of the queried component. It is straightforward to record the signature

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

information for the components in a library. However, using only the exact match of the signatures does not achieve a good recall rate. Relaxed matches need to be defined. The major problem with this method is the precision due to lack of semantic information in the signature. Even with a perfect signature match, there is still no guarantee that the component will be reusable.

Chen et al. use the algebraic specification language (ASL) to specify classes [Che 93]. There are three main sections of a specification: the *sorts* section indicates the data structure of the class; the *opns* section defines the operation names along with their associated input/output parameter types; and the *axioms* section describes the properties of the operations. The sorts and operations symbols constitute the signature of a specification SP, denoted by *sig* (SP). Formally, a signature is a pair $\Sigma = (S, F)$ consisting of a set S of sorts and a set F of operation symbols whereby any operation f $\in$ F is equipped with a functionality which maps a set of sorts $\{s_1, s_2, ..., s_n\}$ to sort s where $s_1, s_2, ..., s_n, s \in S$. When an implementation for a given goal specification SP_g is desired, the repository is searched to find those components whose signatures match $sig(SP_g)$. In addition to an exact match, relaxed matches are determined through renaming and exporting operations upon signatures. The renaming operation defines a map from the sorts and operations of one signature to the other. The exporting operation ignores all sorts and operations in one signature that are not in the other.

Zaremski and Wing [Zar 95a] use Larch/ML, a Larch interface language for the ML programming language, to specify ML functions. The *traits* section of the Larch Shared Language (LSL) introduces *sorts* and *operators* of a function and defines

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

equality between terms composed of the operators and variables of the appropriate sorts. The list of types of the input and output parameters for a function is defined as its signature. Therefore, functional matching based on signature information becomes type matching. Several operations applied to type variables are adapted from [Fie 88]. In addition to exact match, they classify relaxed functional matches as either partial matches or transformation matches. Under partial matches, a generalized match exists when a library type is more general than the query type and a specialized match is the converse of a generalized match. A transformation relaxed match occurs when the library type can be transformed to the query type. The transformation could be curry, uncurry, or reordering. The matching facility is implemented in Emacs.

2.2.2 Facet classification

In the information retrieval area, facet schemes break down information into different categories which make it possible to consider information about several different aspects or properties of the components. Each aspect is denoted as a *facet* [Ran 57]. Prieto-Diaz and Freeman first introduced facet scheme into software reuse practice [Pri 87]. The process of facet classification starts with domain analysis to determine an appropriate set of facets to represent the functionality of the components in the domain of application and all possible or reasonable values for each facet. A thesaurus is derived for each facet to provide vocabulary control. Experienced programmers need to look at each program and provide corresponding values for the facets suggested. Values can only be used within the context of the facet they belong to and ambiguities are resolved through the thesaurus. Conceptual distances between

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

values of each facet are used to evaluate their similarity. Automation only begins after each component is labeled.

The Ithaca Application Development Environment assumes that the target application is object-oriented [Ade 90, Fug 92]. A Software Information Base (SIB), described in the Telos knowledge representation language, collects components of applications previously developed that can be reused. In particular, the SIB is accessible via Selection Tools that are composed of coupled browsing and querying functionality [Fug 93]. The SIB classes are organized according to two facets: VERBs describing the type of services of a class and NOUNs indicating the type of the class upon which the service is performed. Thesauruses are defined for verbs and nouns. A *Functional Description* (FD) is a list of operations describing the functionality of an SIB class. Each operation is composed of three fields: <verb, noun, weight>. The *weight* is a percentage value denoting how well an operation characterizes the behavior of a class. When a class is submitted to SIB, the application developer fills out the associated FD according to his/her understanding of the class along with thesaurus of terms available for facets. A query to SIB consists of two parts: an FD and a threshold value. The query FD is compared to FDs in SIB to compute the *Confidence Value* (CV), a real number ranging between 0 and 1. For any given class in SIB, CV estimates how well it can substitute the query classes. Candidate classes are those that have CVs higher than the given threshold value in the query.

REBOOT (REuse Based on Object-Oriented Techniques) uses four facets to organize classes [Sor 93]. The *Abstraction* facet resembles the class name, i.e. the

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

abstract concept that is implemented by the component. The *operation* facet denotes what methods the component offers. The *operates on* facet describes what other components are cooperating with the classified one. The *dependencies* facet identifies the compilers, operating systems etc. that must be present for the component to work. When a class is inserted to the repository, terms for the four facets are manually recorded. A query to the repository takes the same format, i.e. defines terms for the four facets. The possible terms for one facet are kept in a particular data structure called a *term space*, which is represented by a graph with the vertices being terms and the edges indicating one of three relations between two terms: generalization, specialization and synonym. These relations simplify navigation and allow both exact and relaxed matching candidate components to be selected.

2.2.3 Search by profiles

A profile is a short-form description of a document that plays the role of a surrogate at the retrieval stage. Profiles are easier to manipulate than the entire documents. In reuse repositories, profiles are used to represent the functionality or behavior of the components. Artifacts are selected from the repositories based on the similarities between the query profile and the profiles of artifacts. The classic measurements of similarity in information retrieval are Dice's coefficient, Jaccard's coefficient and Salton's Cosine coefficient [Maa 91].

Maarek et al. adapt an information retrieval approach for automatically constructing software libraries [Maa 91]. Documents are associated with artifacts in software library. For each word-pair, a Lexical Affinity (LA) is computed. LAs of all

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

word-pairs in a given document are used to form the profile of the corresponding artifact. Each document may have a different length of profile. For each document, many potential lexical affinities are defined. But only some of these lexical affinities are conceptually important to the document. The *resolving power* is defined to measure the conceptual importance of a lexical affinity and is based on the assumption that the more frequent a word is in a document, the less information it carries. Only those lexical affinities whose resolving powers are one standard deviation above the mean of all resolving powers contribute to the profile of the document. The similarity between two documents is measured by common lexical affinities in the two associated profiles taking into account the corresponding resolving powers. A method to identify the most useful level clustering is also defined.

2.2.4 Match through verification

With the match through verification approach, searching through a large repository of components is facilitated by specification matching. The assumption behind this method is that a specification, s_i, is associated with each component, p_i, in the library. Specification matching is the process of determining, for a given query q and specification s_i, whether s_i *satisfies* q [Rol 90]. As indicated in Figure 2.3, the specification of a new component is compared with a set of specifications. When a match is found, the implementation of the new component reuses the implementation of the component that is specified by the matched specification. Since the components are specified by formal specifications, a theorem proving mechanism is necessary to show

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

whether an implication exists between the query specification and a given specification in the repository.

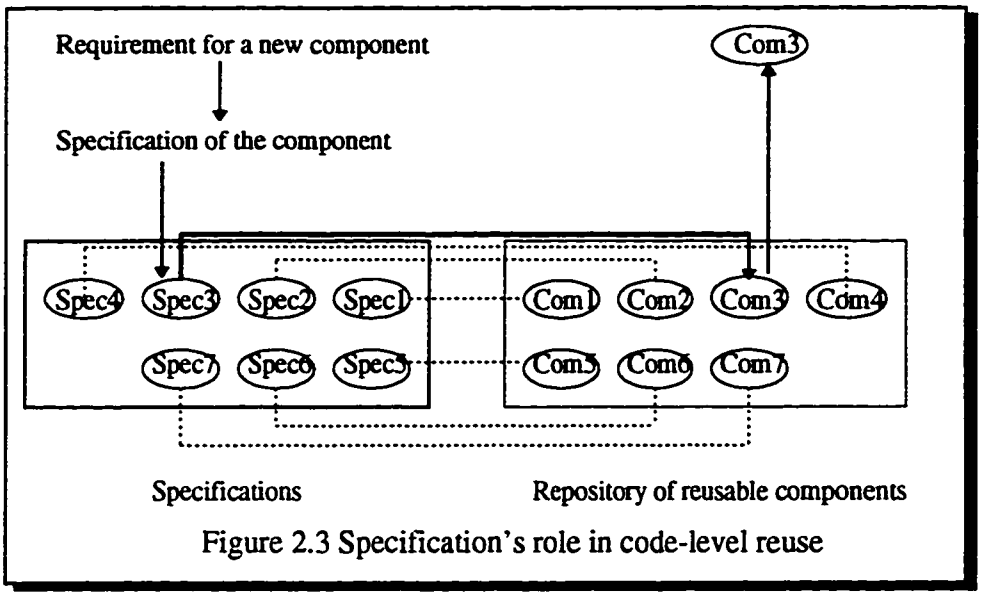

Figure 2.3 Specification's role in code-level reuse

PARIS [Kat 87] supports a library of PARtially Interpreted Schemas. Each schema is accompanied by its *specification* which includes applicability conditions, section conditions, and result assertions. The presentation of a schema includes the text of the schema, the abstract entities set, a statement in temporal logic of all aspects of the schema specification and a proof of correctness of the result assertions of the schema. The proof is conducted assuming both the applicability conditions and the section conditions. A problem statement may have the same form as schema specifications, but will not include free variables or undefined functions. When a user presents a problem statement for a desired program, PARIS searches through the library for a possible candidate schema that could potentially satisfy the user's problem statement. A schema is said to be a possible candidate for the problem statement if there exists a possible substitution of named entities from the schema to the problem

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

statement. In addition, the problem statement needs to imply the schema conditions for that substitution. The result assertion of the schema's specification, when the substitution is made, needs to imply the requirements of the problem statement. If a candidate schema matches the description of requirements from a user, its abstract entities are replaced by concrete entities and the resulting program is presented to the user.

Mili et al. base their approach on formal specifications and the refinement ordering between specifications [Mil 94]. A specification S_p is said to *refine* another specification S_p' if and only if any (programming) solution to S_p is also a solution for S_p'. Given a program, its specification describes functional requirements the program must satisfy using a binary relation that contains all possible combinations of input/output value-pairs that the specifier considers correct. The repository is organized by pairs of <R, p> where p is a program source code and R is a formal specification such that p is correct with respect to R. The query specification K is compared against the specification nodes of the repository to identify those that refine K. The answer set contains the minimal specifications that refine the search argument. Matching of the search argument against a current specification node is formulated as a first order theorem and submitted to the Otter theorem prover.

At an early stage of the Venari project [Win 90], Rollins and Wing propose searching through software libraries using specification matching which is the process of showing an implication holds between the specification associated with a component and the query specification [Rol 91]. A prototype system was implemented using

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

λProlog as the specification and query languages. The specification includes a name, a signature, and a pair of pre- and post-conditions describing the procedure's behavior. The unification feature of λProlog serves as the "free" theorem-proving mechanism. Zaremski extends the relaxation of match among function and defines matching among modules [Zar 95b]. The specifications are written by Larch/ML in terms of pre- and post-condition predicates, denoted by S_{pre} and S_{post} respectively. The specification is interpreted as an implication between the two: $S_{spec} = S_{pre} \Rightarrow S_{post}$. Different degrees of match relaxation are defined based on the combination of the relationship between three corresponding pairs of the query specification and library specification. The determination of match or mismatch relies on theorem proving.

2.3 Summary

This chapter presented a survey on current reuse methodologies from three perspectives: how to populate the repository, the granularity of artifacts, and level of abstraction of the artifacts being reuse. Also four major organization and retrieval methods were discussed. As mentioned in Chapter 1, the challenges for successful reuse include: populating the repository with useful and usable artifacts; determining appropriate representation abstract; easy retrieval; and adaptation guidelines. Most of the research discussed in this chapter bypasses the first challenge and assumes that the artifacts are already collected. On the other hand, even though the needs for modification of the candidate artifacts are widely recognized, few guidelines have been mentioned to assist adaptation of the artifacts retrieved. The emphasis is on representing the artifacts and providing a mechanism for retrieval.

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

A summary of the different approaches presented in this chapter is given in Table 2.1. The first column contains the name of the project or leading author of the proposed method. The second column lists the method employed to populate the repository. The granularity and representation of artifacts are given in the third and fourth columns respectively. The fifth column contains the matching engine of the repository and the last column indicates whether guidelines are provided to assist modification of reusable components. The -- indicates that the aspect is not discussed by the corresponding work.

Table 2.1 Summary of related works

Reuse Project	Method to Populate the repository	Reuse Entity	Representation of artifacts	Matching Engine	Guideline
Castano	Design-for-reuse	Conceptual Model	Key words	Cluster Hierarchy	Yes
SRSGEN	Design-by-reuse	Function	Text	--	--
Esteva	Design-by-reuse	Function	Parameters	--	--
AIRS	--	Function / package	Key words	Graph	--
Hollenbach	--	process	Informal specification	Graph	Yes
Chen	--	Function	Signature	Theorem Prover	--
SIB	--	Object/Class	Facet	Facet Classification	--
REBOOT	--	Object/Class	Facet	Facet Classification	--
Maarek	--	Function	Lexical Affinities	Cluster Hierarchy	--
PARIS	--	Conceptual Model	Temporal Logic	Theorem Prover	Yes
Mill	--	Function	Binary Relation	Theorem Prover	--
VENARI	--	Function / Module	Signature/ Specification	Theorem Prover	--
REST	Design-for-reuse	Transaction/ process	Formal Specification	Database	Yes

The methodology proposed by this research targets all four challenges of successful reuse. RRM makes no assumption of an existing repository but rather

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

defines a repository of reusable frameworks for the software application under development. Thus the repository is built from a design-for-reuse perspective. The frameworks are useful since they are derived from common features of transactions requested by the application. They are also usable since guidelines are provided as part of the frameworks. In the long term, when several systems are developed under this methodology, it will be possible to populate a repository by collecting the frameworks of the developed systems. The granularity of RRM is the transaction level. But the process that provides a systematic way of inferring a repository of reusable frameworks can be adapted and applied to the development of new software systems, thus achieving the granularity of process reuse. Even though our goal is to provide reusability at the implementation stage of software development, the reusable components in the repository are represented by formal specification based on reusability analysis performed at an early stage of the development.

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

Chapter 3

Reusability Analysis Phase

The Requirement Reusability Model is intended to capture the aspect of reuse that a component can be constructed by modifying another component. That is to say that a component, *A*, is reusable by another component, *B*, if *B* can be constructed by modifying *A*. The reusability analysis phase of RRM is intended to infer a clustering structure so that transactions in the same cluster share more functionality, therefore are easier to implement by modifying a transaction in the same cluster rather than a transaction in a different cluster. The requested transactions on a relational database application are specified using the formal specification language Z. The formal specification language is used to specify the functional semantics of transactions and to provide the expressiveness and precision that reuse evaluation requires.

Figure 3.1 shows the main steps and products of the RA phase. First the Z schemas are parsed. A data dictionary and the first version of the transaction dictionary are generated as the result of the parsing step. At this point, the transaction dictionary includes only the IDs of the transactions and the corresponding descriptor sets. Clustering analysis groups the transactions based on their functionality expressed in the

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

descriptor sets. The process produces a cluster structure and extends the transaction dictionary by the corresponding cluster numbers for the transactions.

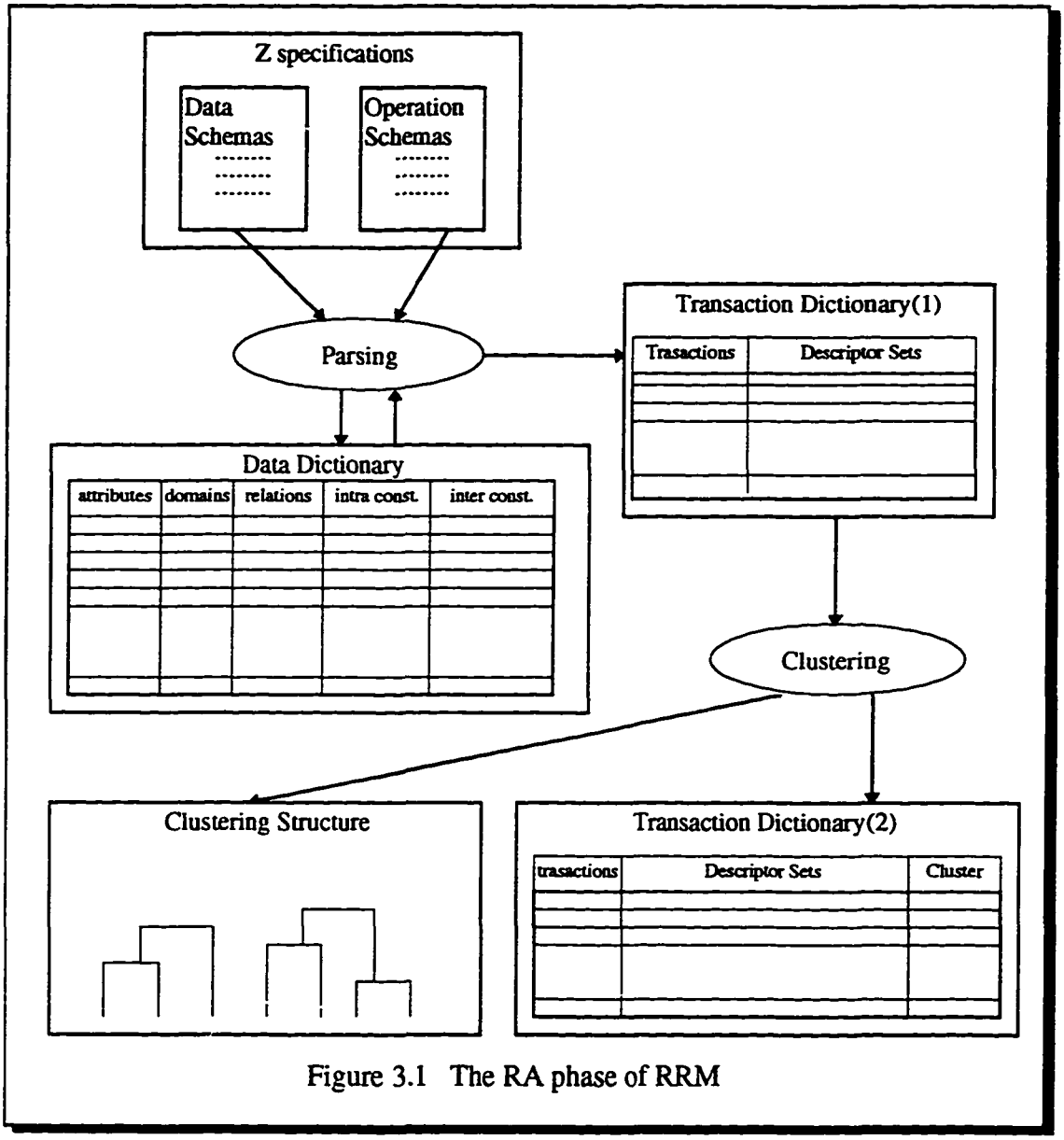

Figure 3.1 The RA phase of RRM

RRM emphasizes that repetitive work exists within the development of a single software system. The inspiration for the emphasis is group technology. Group technology is widely used in manufacture engineering disciplines. It utilizes economies

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

of scope by identifying and exploiting the similarities in the parts to be manufactured and the sequence of machines that are necessary for the processing of those parts [Wem 92]. There are three ways that the benefits of group technology are achieved:

- by performing similar activities together, less time is wasted in changing from one unrelated activity to the next
- by standardizing closely related items or activities, unnecessary duplication of effort is avoided
- by efficiently storing and retrieving information related to recurring problems, search time is reduced.

The reusability analysis phase studies schemas (parts) in order to identify and exploit the similarities in function of the schemas and the sequence of designs that are necessary for the implementation of those schemas. For a large scale software system, the development involves many people where each group may have different expertise. The activity of the design phase can be viewed as allocating portions of the specification to appropriate groups and to appropriate programmers within each group. When similar schemas are allocated to the same group of programmers for implementation, the programmers can concentrate on one type of problem. Therefore, less time is wasted in changing from one unrelated activity to the next even if no explicit reuse is conducted. Engineers in traditional engineering fields are trained to make the best possible choice of artifacts by considering the requirements and selecting an artifact that provides the best balance of parameters. In software engineering practice, design trade-off in relation to requirements and what can be delivered is

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

typically done at requirements analysis and specification time [Pri 96]. Identifying closely related schemas as groups at the specification stage allows specifiers to negotiate with clients and make it possible to design a standardized, reusable framework for one cluster of schemas. The implementation of individual schemas in the cluster can reuse the framework and avoid unnecessary duplication of effort. The third benefit listed above is a by-product of RRM since we keep track of category hierarchy and description information associated with each schema.

One of the means for family identification in group technology is cluster analysis. The goal of clustering is to find groups containing objects that are most homogeneous within these groups, while at the same time the groups are heterogeneous between themselves as much as possible. We employ cluster analysis in the reusability analysis phase to identify families of schemas that require similar sequences of operations. The goal is to infer a classification scheme for the system being developed so that components similar in function are grouped in the same category. Figure 3.2 illustrates the clustering concept. Transactions are described based on features of their functionality.

The success of the reusability analysis phase is based on three important aspects:

1. Rigorous specification by formal language;
2. Measurement space to describe each schema;
3. Similarity indices among schemas.

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

The Hierarchical Agglomerative Clustering (HAC) method is then applied to the similarity matrix with the single linkage method used to calculate the similarity indices among intermediate clusters.

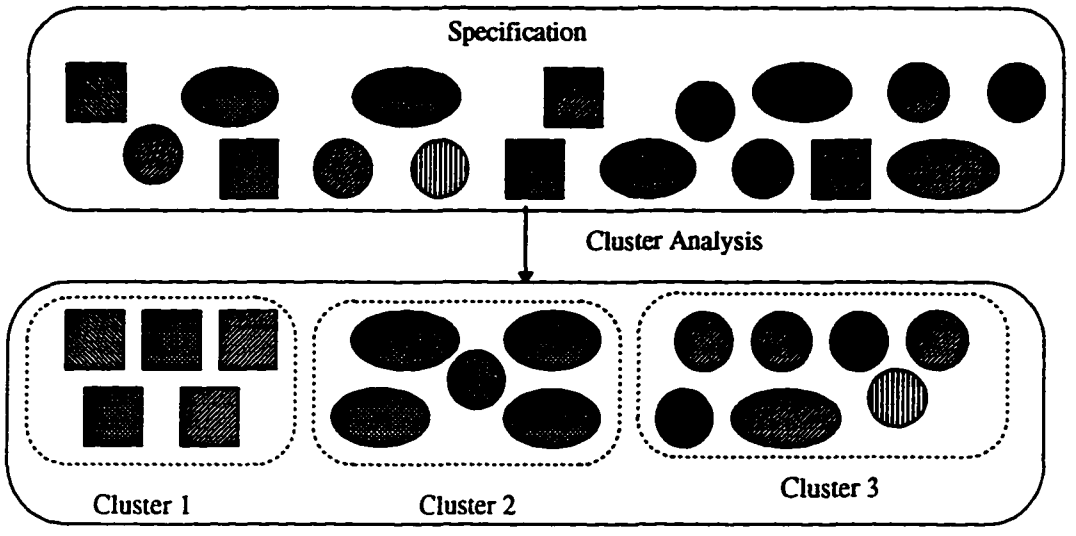

Figure 3.2 Clustering schemas similar in function

3.1 Rigorous Specification of the Transactions

The traditional description of the software life cycle is based on an underlying model, commonly referred as the "waterfall" model[Boe 87]. At the most general level, three phases to the life-cycle are generally agreed upon [Tur 87]:

1. *Analysis phase* which covers the initiation of a project, through the user-needs analysis and feasibility study;
2. *Design phase* which covers the various concepts of system design, broad design, logical design, and physical design;

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

3. *Implementation/maintenance phase* where the computer program is written and tested in terms of verification, validation and sensitivity. When found acceptable, a program is put into use and maintained.

While it is difficult to quantify the effort that goes into each of these phases in general, it is widely acknowledged that a majority of the effort goes into analysis, design, and verification activities [Pre 92]. A specification serves as a contract, a valuable piece of documentation, and a means of communication among a client, a specifier, and an implementer. There is no doubt that the quality of specification will heavily affect the quality of implementation. Applying a formal method to requirement analysis helps clarify a customer's set of informally stated requirements. However, beginners usually find writing formal specifications difficult. Systems, such as KATE [Fic 88], Requirements Apprentice [Reu 91], SPECIFIER [Mir 91], ARIES [Joh 92], etc., have been developed to assist transforming informal requirements into formal specifications.

In a limited domain, a relatively small number of expressions are frequently used to describe certain information [Sol 84]. Soloway and Ehrlich conducted empirical studies of programming knowledge and suggested that expert programmers differ from novice programmers in their use of:

- Programming Plans: Program fragments that represent stereotypic action sequences in programming;
- Rules of Programming Discourse: Rules that specify the conventions in programming.

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

Therefore, with guidelines concerning program plans and rules of discourse available, the difficulty of writing a formal specification should be reduced. At the same time, rigorous formal specification is also critical for function identification. With knowledge of common patterns available, fast and efficient specification processing can be achieved by direct mapping an input schema to its meaning without a full syntactic analysis and without applying conversion rules from the syntactic structure to the semantic interpretation.

In a relational database application, the most common transactions are queries. A query can be used to retrieve a subset of data, summarize data, insert new data, delete unwanted data, and update current data. Queries for the first two purposes are read-only queries that do not modify the contents of the database, while queries for the last three purposes are update queries that do modify the contents of the database. The basic elements in Z, schemas, are used to specify transactions. Patterns for basic operations are listed in Sections 3.1.1 and 3.1.2. These patterns serve as guidelines for specifiers to produce formal specifications.

3.1.1 Read-only operations

Read-only operations are those that do not modify the contents of the database. There are three basic read-only operations in a relational database: select, project, and theta-join. Some common features of basic read-only operations are:

1) include the corresponding Ξ DB schemas;
2) declare input (if any) and output variables;
3) types of output variables are **P**A, where A is a tuple type.

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

They are distinguished from each other by the predicate patterns:

- SELECT: Extracts specified tuples from a specified relation,

  predicate pattern: result! = {tuple : relation | <condition>};

- PROJECT: Extracts specified attributes from a specified relation,

  predicate pattern: result! = {tuple : relation • <attribute_list>};

- THETA-JOIN: Builds a relation consisting of all possible concatenated pairs of tuples, one from each of the two specified relations, such that in each pair the two tuples satisfy some specified condition. The predicate specifies the comparison operation (cop) between values of attributes from the two relations,

  predicate pattern: result! = {t1: rel1 ... tn : reln | ti.att1<cop>tj.att2 $1 \leq i, j \leq n$}.

The Z schema in Figure 3.3 specifies a simple read-only transaction that involves only the *select* operation. The transaction is intended to retrieve full details of those projects that are already finished. A project is finished if the value of its *EndDate* attribute contains a date that is earlier than the current date, i.e. EndDate < Date(). The convention Ξ DB indicates that there is no change of the contents of the database before and after the execution of the transaction.

Proj_End

Ξ DB

lproj! : **P** PROJ

lproj! = {pj : projs | EndDate < Date()}

Figure 3.3 Get full details of all finished projects

Another example of a read-only transaction is intended to retrieve a dataset that contains all project assignments of male employees. The Z schema that specifies the

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

transaction is shown in Figure 3.4. A *select* operation is performed on relation *empls* to retrieve only male employees. A *theta-join* operation is performed on relations *works* and *empls* to get project assignment information.

All_Work

Ξ DB

lwk! : P WORK

lwk! = {wk : works, em : empls | wk.ENum = em.ENum $\wedge$ em.Sex = Male • (wk.ENum, wk.PNum, wk.Hours)}

Figure 3.4 Get all project assignments of male employees

If we are interested in finding IDs of employees who work on the project controlled by the same department that hires them, the Z specification for such a transaction is defined in Figure 3.5. The constraint: em.ENum = wk.ENum helps the transaction to retrieve the projects to which an employee is assigned. The constraint: wk.PNum = pj.PNum makes more details of the project available. Finally, the constraint: pj.CtrlBy = em.HiredBy assures that the department that controls the project hires the employee.

Same_Dept

Ξ DB

lemp! : P ENUM

lemp! = {em : empls, pj : projs, wk : works | em.ENum = wk.ENum $\wedge$ wk.PNum = pj.PNum $\wedge$ pj.CtrlBy = em.HiredBy • (em.ENum)}

Figure 3.5 Employees who work for the projects that are controlled by the same department that hires them.

3.1.2 Update operations

Update operations modify the contents of the database by inserting, updating, and/or deleting tuples of relations. There are basically three kinds of update operations

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

in a relational database: insert, delete and update. Some common features of basic update operations are:

1) include the Δ DB schema;
2) declare input (if any) variables;
3) no output variable;
4) use Ξ DB \ <relations> to specify relations whose contents are changed by the operation.

The specification of the three update operations are distinguished from each other by the predicate patterns:

- Insert a set of tuples to a specified relation

Insert_rel

Δ DB
sr? : **P** REL
Ξ DB \ rel
---
rel' = rel ∪ sr?

- Delete a set of tuples from a specified relation

Delete_rel

Δ DB
sk? : **P** DOM
Ξ DB \ rel
---
rel' = { t : rel | t.dom ∉ sk? }

- Update

Update_rel_att

Δ DB
new? : DOM2
old? : DOM1
Ξ DB \ rel
---
rel' = { t : rel • **if** t.dom1 ∈ old **then** t\(dom2 = new?) **else** t}

A simple insertion transaction is specified using the Z schema in Figure 3.6. The input to the transaction is a tuple of relation *empls*. The insertion operation is specified by the union operation of the set of the original employees and the set that is composed by the

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

input. The convention Δ DB implies changes to the contents of the database and Ξ DB \ empls indicates the changes are upon the relation *empls*.

Inst_em

Δ DB

em? : EMPL

Ξ DB \ empls

empls′ = empls ∪ {em?}

Figure 3.6 Insert a record of employee into the relation

When a transaction is requested to increase the salary of all managers by 2%, the Z schema in Figure 3.7 can be used to specify the transaction. First, the employees, who will be affected by the transaction, are retrieved by joining the relation *empls* and *depts* by the attributes *ENum* and *ManENum*. Then the update of salary occurs to the employees retrieved. Although relation *depts* is involved in the *join* operation, there is no change to the contents of *depts* by the transaction.

Inc_salary

Δ DB

Ξ DB \ empls

let temp = {em : empls, dt : depts | em.ENum = dt.ManENum • em.ENum}

• empls′ = {t : empls • **if** t.ENum ∈ temp **then** t\(salary = salary * 1.02)

else t}

Figure 3.7 Increase the salary of all manager by 2%

The Z schema in Figure 3.8 specifies a transaction that removes all employees that are hired by a given departments. The relation to be affected by the transaction is *empls*. The contents of *empls* after deletion will not include any record with the value of attribute *HiredBy* as 'd?'.

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

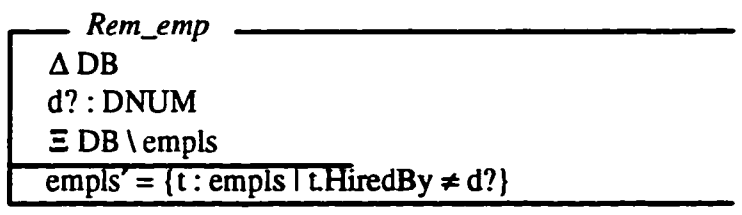

Figure 3.8 Remove all employees hired by department d?

In this section, predicate patterns of basic operations were defined to assist the writing of the formal specification using Z. Most transactions in a relational database application can be composed by a combination of the read-only and update operations listed above. Examples were presented to illustrate the process. This research only considers the basic operations. The advanced operations, i.e. aggregate functions such as *count*, *sum*, *average*, etc., are not discussed in this research.

3.2 Measurement space to describe individual schemas

The specification describes the requirements of an application in an unambiguous way. It works as a contract between the software development group and the client. From the specification, a data dictionary is obtained to store properties of each attribute in the relational database. Entries of the dictionary are attributes of relations including the intra and inter relations constraints applicable to the attributes . Such information is recorded because such properties of attributes may affect the level of effort required for the transaction.

This subsection defines a measurement space so that each schema can be compared at the same level of scale. Each transaction is described by a descriptor set defined in section 3.2.1. Intersection and union operations between two descriptor sets

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

are defined in sections 3.2.2 and 3.2.3 respectively. In section 3.2.4, the cardinality of a descriptor set is defined to assist further comparison among transactions.

3.2.1 Descriptor sets

Z schemas that specify transactions according to the guidelines provided in section 3.1 are suitable for automatic indexing since they are structured at both syntactic and semantic levels and belong to a well identifiable domain. The goal of this analysis phase is to identify families of schemas that require similar sequence of actions. Therefore, the measurement space needs to reflect information of the types of basic operations involved in the transaction and level of efforts required for the operations. Each transaction of a relational database application is measured by a set of <descriptor, value> pairs. The set is call a *descriptor set*. Each *descriptor* indicates one type of basic operation requested by the transaction and the associated *value* indicates the level of effort required for the operation.

There are seven possible <descriptor, value> pairs, each identifies a basic operation involved in a transaction and the level of effort for the operation. They are listed as follows:

- *<Input, [value | set]>*: Depending on the purpose of transaction, an input interface may be required to allow a transaction to select, delete, insert or update data with or to special value(s). For instance, a transaction may want to retrieve information of all employees working for a given department. In this case, <input, value> is a member of the descriptor set of the transaction. On the other hand, a transaction may want to move a list of projects from one department to another. In this case,

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

<input, set> is a member of the descriptor set of the transaction. We distinguish the two situations because the levels of effort required for building an input interface for a single value and a set of values are different.

- *<Output, [old | new]>*: The output of a query in a relational database is always a relation. However, when the purpose of a transaction is to modify the contents of an existing relation, no explicit output is declared. For example the transaction shown in Figure 3.6 does not produce any explicit output. In this case, the descriptor set for the transaction does not include any pair with "output" as the descriptor. On the other hand, the resulting relation may either take the structure of an existing relation or require a new structure. The structure of the resulting relation of a given transaction can be determined both by whether a project operation occurs and the declaration of the output variable. The transaction shown in Figure 3.3 produces a relation that takes the structure of original project relation. In this case, <output, old> is a member of the descriptor set of the transaction. If we are only interested in the dates those projects are finished, we will need to construct a new structure to record such information. In this case, <output, new> is a member of the descriptor set of the transaction.
- *<Select, N>*: The *select* operation applies a filter to a relation to choose a subset for further actions. When more than one relation is involved in the transaction, filters may be applied to different relations. The value of N indicates the number of relations that will be filtered. For example, a transaction may want to know all

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

female employees who work for projects located in New York. In this case, <select, 2> is a member of the descriptor set for the transaction.

- *<Join, N>*: The *join* operation links two or more relations together based on certain criteria. The value of N indicates the number of relations that are linked together by the *join* operation. If a transaction is interested in all employees who work for a given project, relations *empls* and *works* need to be linked together based on common projects number. In this case, <join 2> is a member of the descriptor set of the transaction.
- *<Delete, N>*: The *delete* operation deletes record(s) from a relation. We consider the case of delete restricted, i.e., the deletion of any primary key value is only allowed when there is no foreign key with the same value. For example, we would not want to delete an employee record when he/she is still the manager of a department. The value of N indicates the number of relations having a foreign key referring to the primary key of the relation whose record(s) are to be deleted.
- *<Insert, N>*: The *insert* operation inserts record(s) into a relation. As required by the data integrity rule of the relation database, no foreign key is allowed to have values that are not values of the corresponding primary key. For instance, we do not want to insert any record of employee while the department that hires him/her does not exist in the database. The value of N indicates the number of relations whose primary key corresponds to attributes in the relation to which the new records are to be inserted.

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

- *<Update, (M, N)>*: The *update* operation changes value(s) of record(s) in a relation. We need to consider both cases of changing a primary key value and changing a foreign key value. The value of M indicates the number of relations whose primary key corresponds to attributes affected by the operation and the value of N indicates the number of relations having foreign key referring to the primary key when affected by the operation.

By searching for the patterns defined in section 3.1 with the assistance of knowledge of primary/foreign keys stored in the data dictionary, each transaction can therefore be described by a set of <descriptor, value> pairs. Not every transaction requires the same kind of basic operations. For example, the example transaction in Figure 3.3 only requires a *select* operation which retrieves a subset of the contents in relation *projs*, while the example transaction in Figure 3.4 requires a *join* operation to link two relations together. Therefore, the sizes of descriptor sets for transactions may differ from each other.

In order to illustrate the process of extracting a descriptor set from a given schema, three more example transactions are listed below. Discussions are given to explain the process.

The Z schema in Figure 3.9 specifies a transaction that retrieves the IDs for the departments whose manager is a given person.

Dept_m

Ξ DB
m? : ENUM
ldpt! : **P** DNUM
ldpt! = {dpt : depts | ManENum = m? • DNum}

Figure 3.9 Get IDs of all departments whose manager is m?

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

In the schema, m is decorated with ? and ENUM is a singular type not a set, therefore, <input, value> is included in its descriptor set. "ldpt" is decorated with ! and "**P** DNUM" indicate the structure of the resulting relation, therefore, <output, new> is included in its descriptor set. The other pair of <descriptor, value> in the descriptor set is <select, 1> since "ldpt! = {dpt : depts | ManENum = m? . . .} fits the description of the pattern of *select* operation with one relation involved.

When a transaction is intended to collect working hours of a given set of employees, the Z schema can be defined as shown in Figure 3.10. In the schema, "sen" is decorated by ? and **P** ENUM is used to indicate a set of values is required, therefore, <input, set> is included in its descriptor set. Since "lwk!" takes the structure of "WORK", an existing relation, <output, old> is included in its descriptor set. The other <descriptor, value> pair detected is <join, 2> since the predicate matches the pattern of *join* operation: result! = {t1:rel1, t2 :rel2 ... | t1.att1 <cop> t2.att2 ...} with two relations involved.

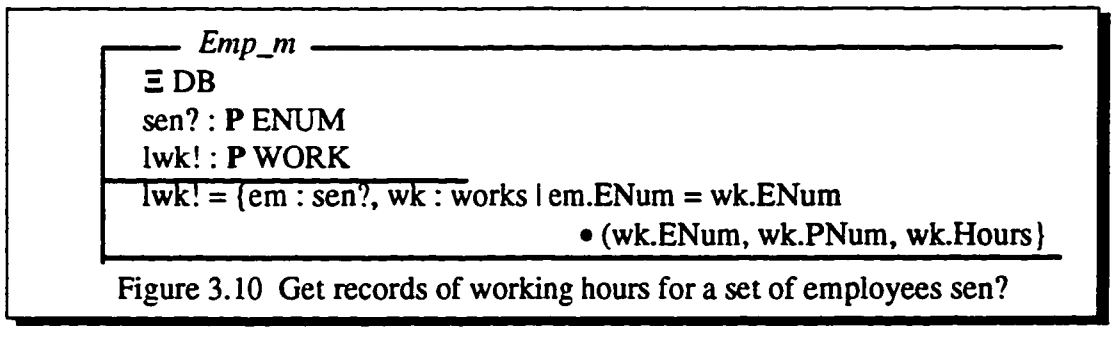

Figure 3.10 Get records of working hours for a set of employees sen?

Assume that the HiredBy attribute (the department number by which the employee is hired) in relation EMPL is the foreign key corresponding to DNum attribute in relation DEPT. We would like to make sure that when inserting a new

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

employee record to the relation EMPL, the HiredBy value in the record should be a value exists in relation DEPT. The schema in Figure 3.11 indicates the transaction of inserting a set of employee records.

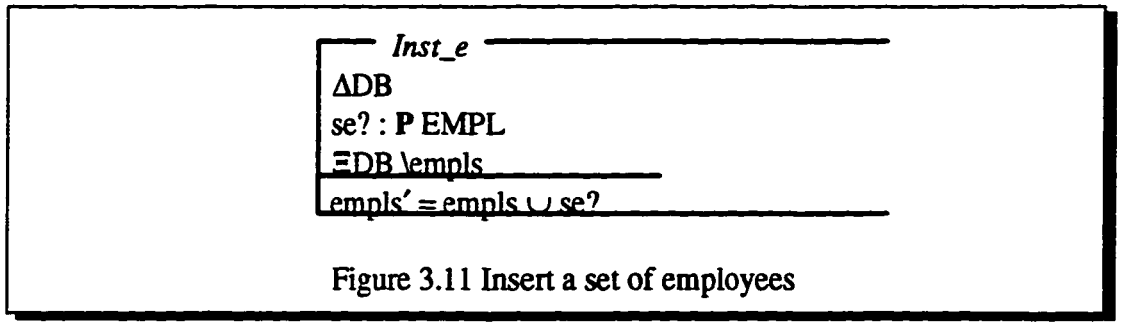

Figure 3.11 Insert a set of employees

Such a schema will have <insert, 1> in its descriptor set since there is one other relation whose primary key corresponds to one attribute in se?.

The descriptor sets for schemas shown in Figures 3.3 through 3.11 are summarized in Table 3.1:

Table 3.1 Descriptor sets

Figure	Descriptor Sets
3.3	<output, old>, <select, 1>
3.4	<output, old>, <select, 1>, <join, 2>
3.5	<output, new>, <join, 3>
3.6	<input, value>, <insert, 1>
3.7	<output, new>, <join, 2>, <update, (0, 0)>
3.8	<input, value>, <delete, 2>
3.9	<input, value>, <output, new>, <select, 1>
3.10	<input, set>, <output, old>, <join, 2>
3.11	<input, set>, <insert, 1>

3.2.2 Intersection of two descriptor sets

We define the intersection of two descriptor sets as another descriptor set. The algorithm in Figure 3.12 computes the intersection of two descriptor sets. The algorithm first determines the common type of operations involved in the two

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

transactions. If there is no common type of operation in the two transactions, the resulting intersection is an empty set. When such an operation exists, the algorithm then determines the minimum effort required by the two transactions based on corresponding values of the <descriptor, value> pairs. If the common operation is either *input* or *output*, whether such descriptor will be part of the resulted intersection depends on whether the operations in the two transactions involve the same level of effort. If, for example, both transactions request an input interface to a get value, <*input*, value> is included in the resulting intersection. If not, the operation is ignored by the resulting intersection. Since the values associated with operations *select*, *join*, *delete*, *insert* and *update* indicate the number of relations involved in the operations, the minimum effort required by the two transactions is computed by the smaller value of the two <descriptor, value> pairs. Based on the algorithm, intersections among the descriptor sets in Table 3.1 are computed and listed in Table 3.2.

```
Intersect = ∅;
For each op_ik = op_jl do
   If (op_ik = "input" or op_ik = "output") and v_ik = v_jl
      then  Intersect = Intersect ∪ {< op_ik, v_ik>};
   If op_ik is in {"select", "join", "delete", "insert"}
      then  Intersect = Intersect ∪ {< op_ik, min{ v_ik , v_jl }>};
   If op_ik = "update"
      then  Intersect =Intersect ∪ {<op_ik, (min{v_ik1,v_jl1}, min{v_ik2,v_jl2 })>};
```

Figure 3.12 Intersection algorithm

3.2.3 Union of two descriptor sets

The union of two descriptor sets is also defined as a descriptor set. The algorithm in Figure 3.13 computes the union of two descriptor sets. The purpose of the

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

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

Table 3.2 Pair-wise intersection among transactions

Fig.	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
3.2	<output, old> <select, 1>	<output, old> <select, 1>	Ø	Ø	Ø	Ø	<select, 1>	<output, old>	Ø
3.3	<output, old> <select, 1>	<output, old> <select, 1> <join, 2>	<join, 2>	Ø	<join, 2>	Ø	<select, 1>	<join, 2>	Ø
3.4	Ø	<join, 2>	<output, new> <join, 3>	Ø	<output,new> <join, 2>	<output,new>	<output, new>	<join, 2>	Ø
3.5	Ø	Ø	Ø	<input,value> <insert, 1>	Ø	<input,value>	<input, value>	Ø	<insert, 1>
3.6	Ø	<join, 2>	<output, new> <join, 2>	Ø	<output,new> <join, 2> <update,(0,0)>	<output, new>	<join, 2>	<join, 2>	Ø
3.7	Ø	Ø	<output, new>	<input,value>	<output, new>	<input, value> <output, new> <delete, 2>	<input, value> <output, new>	Ø	Ø
3.8	<select, 1>	<select, 1>	<output, new>	<input,value>	<join, 2>	<input, value> <output, new>	<input, value> <output, new> <select, 1>	Ø	Ø
3.9	<output, old>	<output, old> <join, 2>	<join, 2>	Ø		Ø	Ø	<input, set> <output, old> <join, 2>	<input, set>
3.10	Ø	Ø	Ø	<insert, 1>	Ø	Ø	Ø	<input, set>	<input, set> <insert, 1>

algorithm defined in Figure 3.13 is to identify the operations and the corresponding effort necessary to accomplish both transactions. When the two descriptor sets are identical, the resulting union takes the same descriptor set. The algorithm first searches for the common type of operations requested by the two transactions. When such an operation exists, the algorithm then determines the maximum effort required by the two transactions based on corresponding values of the <descriptor, value> pairs.

```
Union = ∅;
For each  op_ik = op_jl  do
   If (op_ik = "input" or op_ik = "output")
       if (v_ik ≠ v_jl )
          then   Union = Union ∪ {< op_ik, ?>}
          else   Union = Union ∪ {< op_ik, v_ik >};
   If op_ik is in {"select", "join", "delete", "insert"}
       then  Union = Union ∪ {<op_ik, max{ v_ik , v_jl }>};
   If op_ik = "update"
       then  Union = Union ∪ {<op_ik, (max{ v_ik1 , v_jl1 }, max{ v_ik2 , v_jl2 })>};
For each op_ik that does not have matching op_jl do
   If op_ik = "update"
       then  Union = Union ∪ {<op_ik, (v_ik1, v_ik2)>}
       else  Union = Union ∪ {<op_ik, v_ik >};
For each op_jl that does not have matching op_ik do
   If op_jl = "update"
       then  Union = Union ∪ {<op_jl, (v_jl1, v_jl2)>}
       else  Union = Union ∪ {<op_jl, v_jl >};
```

Figure 3.13 Union algorithm

If the common operation is either *input* or *output*, the algorithm searches further to determine whether the operations in the two transactions involve the same level of effort. If, for example, both transactions request an output interface to build a relation that takes the structure of an existing relation, *<output*, old> is included in the resulting union. If not, a question mark '?' is the value associated with the descriptor

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

for the resulting union indicating the two transactions require different output interfaces. Since the values associated with operations *select, join, delete, insert* and *update* indicate the number of relations involved in the operations, the maximum effort required by the two transactions is computed by the larger value of the two <descriptor, value> pairs. For operations only requested by one transaction, the union contains the respective <descriptor, value> pair. Using the Union Algorithm, we determine the union between all pairs of descriptor sets that are listed in Table 3.1. The results are shown in Table 3.3.

3.2.4 Cardinality of a descriptor set

The values in descriptor sets measure the level of effort associated with basic operations required by the transaction. For *input* and *output* operations, the value reflects the type of interface requested by a given transaction. For *select* and *join* operations, the value indicates the number of relations involved in the operations. The value associated with *insert, delete*, and *update* depends on the number of intra or inter relation constraints that may be violated by the operations. The cardinality of a descriptor set is defined to reflect the total effort of all basic operations involved in each transaction. The algorithm presented in Figure 3.14 computes the cardinality of a descriptor set. For each basic operation, a value E_{op} is assigned to indicate the unit effort that is necessary to implement the operation. E_{inp} indicates the unit effort to set up an input interface. E_{out} indicates the unit effort to declare the structure of the resulting relation and to set up an output interface for displaying the result. E_{sel} measures the effort to apply filter to one relation. E_{join} measures the effort to link two

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

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

Table 3.3 Pair-wise union among transactions

Fig.	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
3.2	<output, old> <select, 1>	<output, old> <select, 1> <join, 2>	<output, ?> <select, 1> <join, 3>	<input,value> <output, old> <select, 1> <insert, 1>	<output, ?> <select, 1> <join, 2> <update,(0,0)>	<input, value> <output, ?> <select, 1> <delete, 2>	<input, value> <output, ?> <select, 1>	<input, set> <output, old> <select, 1> <join, 2>	<input, set> <output, old> <select, 1> <insert, 1>
3.3	<output, old> <select, 1> <join, 2>	<output, old> <select, 1> <join, 2>	<output, ?> <select, 1> <join, 3>	<input,value> <output, old> <select, 1> <join, 2> <insert, 1>	<output, ?> <select, 1> <join, 2> <update,(0,0)>	<input, value> <output, ?> <select, 1> <join, 2> <delete, 2>	<input, value> <output, ?> <select, 1> <join, 2>	<input, set> <output, old> <select, 1> <join, 2>	<input, set> <output, old> <select, 1> <join, 2> <insert, 1>
3.4	<output, ?> <select, 1> <join, 3>	<output, ?> <select, 1> <join, 3>	<output, new> <join, 3>	<input,value> <output,new> <join, 3> <insert, 1>	<output, new> <join, 3> <update,(0,0)>	<input, value> <output, new> <join, 3> <delete, 2>	<input, value> <output, new> <select, 1> <join, 3>	<input, set> <output, ?> <join, 3>	<input, set> <output,new> <join, 3> <insert, 1>
3.5	<input,value> <output, old> <select, 1> <insert, 1>	<input,value> <output, old> <select, 1> <join, 2> <insert, 1>	<input, value> <output, new> <join, 3> <insert, 1>	<input,value> <insert, 1>	<input, value> <output, new> <join, 2> <insert, 1> <update,(0,0)>	<input, value> <output, new> <insert, 1> <delete, 2>	<input, value> <output, new> <select, 1> <insert, 1>	<input, ?> <output, old> <join, 2> <insert, 1>	<input, ?> <insert, 1>
3.6	<output, ?> <select, 1> <join, 2> <update,(0,0)>	<output, ?> <select, 1> <join, 2> <update,(0,0)>	<output, new> <join, 3> <update, (0,0)>	<input,value> <output,new> <join, 2> <insert, 1> <update,(0,0)>	<output, new> <join, 2> <update,(0,0)>	<input, value> <output, new> <join, 2> <delete, 2> <update,(0,0)>	<input, value> <output, new> <select, 1> <join, 2> <update,(0,0)>	<input, set> <output, ?> <join, 2> <update,(0,0)>	<input, set> <output, new> <join, 2> <insert, 1> <update, (0,0)>
3.7	<input, value> <output, ?> <select, 1> <delete, 2>	<input, value> <output, ?> <select, 1> <join, 2> <delete, 2>	<input, value> <output, new> <join, 3> <delete, 2>	<input, value> <output, new> <insert, 1> <delete, 2>	<input, value> <output, new> <join, 2> <delete, 2> <update, (0,0)>	<input, value> <output, new> <delete, 2>	<input, value> <output, new> <select, 1> <delete, 2>	<input, ?> <output, ?> <join, 2> <delete, 2>	<input, ?> <output, new> <insert, 1> <delete, 2>

(table con'd.)

	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
3.8	<input, value> <output, ?> <select, 1>	<input, value> <output, ?> <select, 1> <join, 2>	<input, value> <output, new> <select, 1> <join, 3>	<input, value> <output, new> <select, 1> <insert, 1>	<input, value> <output, new> <select, 1> <join, 2> <update,(0,0)>	<input, value> <output, new> <select, 1> <delete, 2>	<input, value> <output, new> <select, 1>	<input, ?> <output, ?> <select, 1> <join, 2>	<input, value> <output, new> <select, 1> <insert, 1>
3.9	<input, set> <output, old> <select, 1> <join, 2>	<input, set> <output, old> <select, 1> <join, 2>	<input, set> <output, ?> <join, 3>	<input, ?> <output, old> <join, 2> <insert, 1>	<input, set> <output, ?> <join, 2> <update,(0,0)>	<input, ?> <output, ?> <join, 2> <delete, 2>	<input, ?> <output, ?> <select, 1> <join, 2>	<input, set> <output, old> <join, 2>	<input, set> <output, old> <join, 2> <insert, 1>
3.10	<input, set> <output, old> <select, 1> <insert, 1>	<input, set> <output, old> <select, 1> <join, 2> <insert, 1>	<input, set> <output, new> <join, 3> <insert, 1>	<input, ?> <insert, 1>	<input, set> <output, new> <join, 2> <insert, 1> <update,(0,0)>	<input, ?> <output, new> <insert, 1> <delete, 2>	<input, value> <output, new> <select, 1> <insert, 1>	<input, set> <output, old> <join, 2> <insert, 1>	<input, set> <insert, 1>

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

relations together. E_{ins} is the unit effort of inserting a record into a relation. E_{del} reflects the effort of deleting a record from a relation. E_{fk} checks whether a foreign key constraint is violated and E_{pk} checks whether a primary key constraint is violated.

```
Cardinality = 0;
For each op_ik do
   If (op_ik = "input") then
      if (v_ik = '?')
      then  Cardinality = Cardinality + 2 * E_inp
      else  Cardinality = Cardinality + 1 * E_inp;
   If (op_ik = "output") then
      if (v_ik = '?')
      then  Cardinality = Cardinality + 2 * E_out
      else  Cardinality = Cardinality + 1 * E_out;
   If (op_ik = "select")
      then  Cardinality = Cardinality + v_ik * E_sel;
   If (op_ik = "join")
      then  Cardinality = Cardinality + v_ik * E_join;
   If (op_ik = "insert")
      then  Cardinality = Cardinality + v_ik * E_pk + E_ins;
   If (op_ik = "delete")
      then  Cardinality = Cardinality + v_ik * E_fk + E_del;
   If op_ik = "update"
      then Cardinality = Cardinality + v_ik1 * E_ins + v_jl2 * E_del + E_upd;
```

Figure 3.14 Cardinality algorithm

For *input* and *output* operations, one type of interface need to be built when the associated values are determined. Thus the cardinality is increased by one unit of effort when such <descriptor, value> exists. However, since the descriptor set may be generated by the union algorithm of two transactions, the associated value may not be determined. In such a case, the interface needs to consider both possibilities of the input being a value or a set, or output taking an existing or non-existing structure. Therefore, the cardinality is increased by two units of effort. For *select* and *join* operations, the associated values in the descriptor set indicate the number of relations involved in the

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

operations. Thus, the cardinality is increased by the value times the unit effort, E_{sel} and E_{join}, respectively. The contribution from *insert*, *delete*, and *update* operations is determined by the number of intra and inter relation constraints that may be affected by the operations. Tables 3.4 and 3.5 show cardinalities of pair-wise intersections and unions among transactions in Table 3.1, assuming the unit effort for all operations is the same.

Table 3.4 Cardinalities of pair-wise intersections

Figure	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11
3.3	2	2	0	0	0	0	1	1	0
3.4	2	4	2	0	2	0	1	3	0
3.5	0	2	4	0	3	1	1	2	0
3.6	0	0	0	3	0	1	1	0	2
3.7	0	2	3	0	4	1	2	2	0
3.8	0	0	1	1	1	5	2	0	0
3.9	1	1	1	1	2	2	3	0	0
3.10	1	3	2	0	2	0	0	4	1
3.11	0	0	0	2	0	0	0	1	3

Table 3.5 Cardinalities of pair-wise unions

Figure	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11
3.3	2	4	6	5	6	7	4	5	5
3.4	4	4	6	7	6	9	6	5	7
3.5	6	6	4	7	5	8	6	6	7
3.6	5	7	7	3	7	7	5	7	4
3.7	6	6	5	7	4	8	6	6	7
3.8	7	9	8	7	8	5	6	9	8
3.9	4	6	6	5	6	6	3	7	6
3.10	5	5	6	7	6	9	7	4	6
3.11	5	7	7	4	7	8	6	6	3

3.3 Clustering Analysis

Classification is a statistical technique concerned with separating distinct sets of objects and allocating new objects to previously defined groups. It pertains to a known

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

number of groups. The emphasis is on deriving a rule that can be used to optimally assigned a new object to the respective labeled classes. Different classification approaches have been proposed for organizing software collections into a software library system to facilitate query and retrieval [Dev 90, Jon 88]. Cluster analysis, on the other hand, makes no assumption of an existing structure of the repository, but rather infers a classification scheme based on available observations. Clusters are groups of objects linked together according to a set of rules. The goal of clustering is to find groups containing objects most homogeneous within these groups, while at the same time maximizing the heterogeneity among the groups.

3.3.1 Similarity function

The term "object" is used in a very broad sense. An object can be anything that could be represented as a point in a multidimensional measurement space. The purpose of the clustering in the reusability analysis phase is to group together the transactions that are more homogeneous in function. In this work, transactions are objects of interest. Each transaction is formally specified by a Z schema. It is important that all objects that are to be clustered are defined (measured) in the same measurement space. Section 3.2 defines the measurement space of individual transactions including the descriptor sets of transactions and the set operations *intersection*, *union* and *cardinality*. Once the objects are defined, two functions representing the relationships among the objects need to be specified. These functions are the "similarity" function and "distance" function, with one function inversely proportional to the other. A non-negative real function $F(O_i, O_j)$ is said to be a similarity measure if

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

1) $0 \le F(O_i, O_j) \le 1$ for all $i <> j$

2) $F(O_i, O_i) = 1$

3) $F(O_i, O_j) = F(O_j, O_i)$

while a distance function $D(O_i, O_j)$ is defined so that

1) $D(O_i, O_j) >= 0$ for all i , j

2) $D(O_i, O_i) = 0$

3) $D(O_i, O_j) = D(O_j, O_i)$

These functions are used as basic criteria to determine clusters. Since it is always possible to substitute at the end of the clustering procedure the "distance" with the reverse "similarity" or vice versa, we concentrate on only one function at the time of analysis. In this research, the similarity indices are used to indicate the common efforts involved in two transactions.

Let $d_i = \{<op_{i1}, v_{i1}>, <op_{i2}, v_{i2}>, ..., <op_{im}, v_{im}>\}$ be the descriptor set for schema s_i and $d_j = \{<op_{j1}, v_{j1}>, <op_{j2}, v_{j2}>, ..., <op_{jn}, v_{jn}>\}$ be the descriptor set for schema s_j, where m and n are the number of operations respectively. The similarity index of the two schemas is determined by the percentage of common descriptors (Jaccard's coefficient): $S(s_i, s_j) = \dfrac{|d_i \cap d_j|}{|d_i \cup d_j|}$ with special definition of the intersection and union of the two descriptor sets and the cardinality of a descriptor set as defined in sections 3.2.2, 3.2.3, and 3.2.4, respectively.

For any two schemas, s_i and s_j, the following cases can occur:

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

- $S(s_i, s_j) = 0$ -- the two schemas do not share any common basic operations.
- $0 < S(s_i, s_j) < 1$ -- the two schemas share some common basic operations. A special case is the one schema may contain all basic operations the other has.
- $S(s_i, s_j) = 1$ -- the two schemas have no basic operations other than the common ones. They perform identical transactions but may be unto different relations.

Based on the cardinalities computed in Tables 3.4 and 3.5, the matrix of pair-wise similarity indices among descriptor sets in Table 3.1 is listed in Table 3.6. Since the example transactions are used to illustrate how different functionality can be accomplished, the percentage of common functionality between any two transactions is low. Therefore, the similarity indices in Table 3.6 are low.

Table 3.6 Similarity matrix

Figure	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11
3.3	1.00	0.50	0.00	0.00	0.00	0.00	0.25	0.20	0.00
3.4	0.50	1.00	0.33	0.00	0.33	0.00	0.17	0.60	0.00
3.5	0.00	0.33	1.00	0.00	0.60	0.12	0.17	0.33	0.00
3.6	0.00	0.00	0.00	1.00	0.00	0.14	0.20	0.00	0.50
3.7	0.00	0.33	0.60	0.00	1.00	0.12	0.17	0.33	0.00
3.8	0.00	0.00	0.12	0.14	0.12	1.00	0.33	0.00	0.00
3.9	0.25	0.17	0.17	0.20	0.17	0.33	1.00	0.00	0.00
3.10	0.20	0.60	0.33	0.00	0.33	0.00	0.00	1.00	0.17
3.11	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.17	1.00

3.3.2 Hierarchical agglomerative clustering

There is no single definition of a cluster, but it is generally agreed that a cluster is a group of objects whose members are more similar to each other than to the members of any other group. Clusters should be represented in the measurement space in the same form as the objects to allow identical treatment for clusters and objects.

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

Therefore, we can extend our measurement functions to apply to clusters of objects. The Hierarchical Agglomerative Clustering (HAC) method [Tur 90] iteratively builds a sequence of disjoint clusters covering the original set of objects. The algorithm is described in Figure 3.15.

```
Let c_I = { s_i } for 1 ≤ i = I ≤ n
While # of clusters > 1 do
    Begin
       For all pairs of cluster I and J
              compute S_I,J;
       Search for I and J such that S_I,J is maximum
              c_I = c_I ∪ c_J ;
              Delete c_J ;
    End;
```

Figure 3.15 Traditional HAC method

The clustering process starts with the clusters of singleton objects and identifies the two clusters that are most closely related, i.e. have the largest similarity index. It then merges them into a single cluster. At each phase, similarity indices among clusters are updated due to merges. Therefore, a similarity measurement between two clusters needs to be defined. It is derived from the similarity measurement between two transactions based on the single linkage method:

$$S_{c_I,c_J} = \min\{S(s_i,s_j) | s_i \in c_I, s_j \in c_J\}$$

i.e. the similarity index of two clusters is the minimum similarity index among similarity indices of all possible cross-cluster pairs of transactions.

The purpose of the clustering analysis in this work is to identify families of transactions that are similar in function for the potential of reuse. The traditional HAC procedure (Figure 3.14) is repeated until only one cluster remains. Since we only

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

design with reuse when adaptation cost is less than the cost of developing the component from scratch, we are only interested in grouping two transactions into one cluster when the percentage of common functionality exceeds 50%. Therefore, the HAC procedure stops when the similarity index chosen, i.e. the maximum efforts one would like to invest on adaptation rather than design from scratch, falls below 0.6. An updated algorithm of HAC method presented in Figure 3.15 is listed as follows:

```
Let cI = { si } for 1 ≤ i = I ≤ n
While max{ SI,J } > 0.6 do
    Begin
        For all pairs of cluster I and J
                compute SI,J;
        Search for I and J such that SI,J is maximum
                cI = cI ∪ cJ ;
                Delete cJ ;
    End;
```

Figure 3.16 Clustering algorithm

The process starts with the clusters of singleton objects and iteratively searches and merges the two clusters that are more similar to each other than the rest of the clusters. The only difference is that the process terminates before all transactions are grouped into a single cluster. Instead, the process completes when the maximum value of the similarities among all remaining clusters drops below the threshold value 0.6.

3.4 Summary

This chapter presented a clustering analysis approach to perform reusability analysis on formal specifications of transactions in relational database applications. The purpose is to identify repetitive works within a single software application.

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

Transactions are specified using Z. A measurement space is defined to represent individual transactions. Each transaction is described as a set of <descriptor, value> pairs where possible descriptors are basic operations requested in the transaction and values indicate the level of effort. The intersection and union of two descriptor sets are defined and algorithms are given to assist evaluation. A unary operation is defined to compute the cardinality of a descriptor set. The descriptor sets are extracted from Z schemas of the transactions and then used to compute the similarity indices among transactions. A similarity function is used as the basic criteria for the clustering analysis. Similarity indices among transactions are determined by a revised version of Jaccard's coefficient. Similarity indices among intermediate clusters are derived based on the single linkage method. The cluster hierarchy is generated by the HAC method. A threshold value is given to preserve the usefulness of the clusters.

In Chapter 4, we describe the reusability realization phase which is the next phase of RRM. The resulting clusters along with the descriptions of the transactions serve as input to determine the generic requirements for each cluster. From these clusters, frameworks are generated as reusable components for use in the implementation of individual transactions.

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

Chapter 4

Reusability Realization Phase

The RA phase of RRM, as described in the previous chapter, generates a classification scheme of the repository based on the formal specification of transactions. A cluster structure is derived where transactions that are similar in function are grouped into the same clusters. The RR phase of RRM continues the process by generating reusable frameworks from the analysis of the descriptor sets associated with the transactions within clusters. The implementation of individual transactions reuses the frameworks. The RR phase can be divided into two parts. The first part, as illustrated in Figure 4.1, extracts the generic requirements for clusters based on the descriptor sets of the transactions within the clusters. Then frameworks are generated to accomplish the generic requirements extracted. The frameworks provide the templates for the implementation of individual transactions. Section 4.1 defines algorithms to extract generic requirements of each cluster based on descriptor sets of member transactions of the cluster. The requirements are expressed in the same measurement space as individual transactions using descriptor sets. Frameworks that accomplish these generic requirements are generated automatically as described in section 4.2. Each framework is composed of three parts: interface, local variables and body of the framework.

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

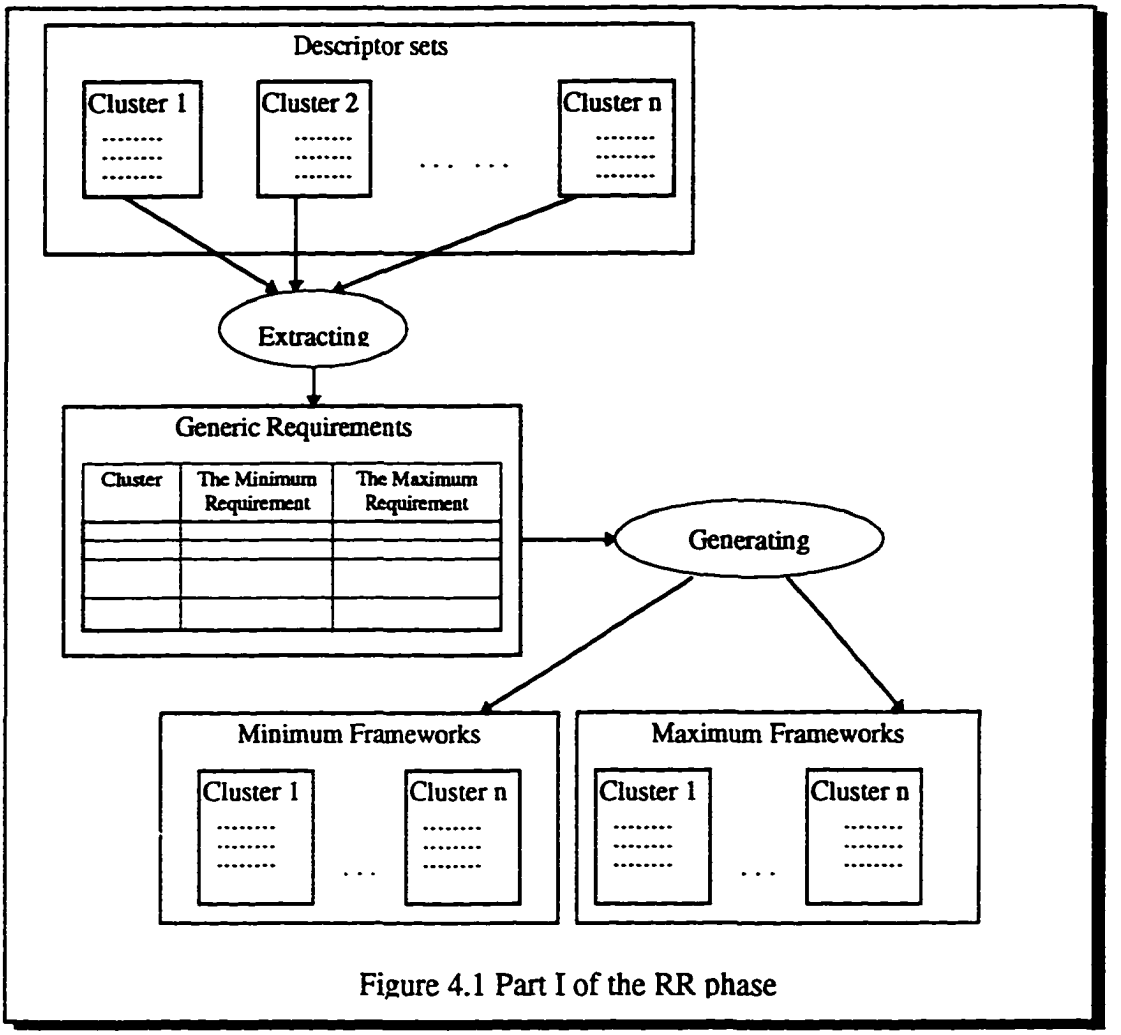

Figure 4.1 Part I of the RR phase

The second part of the RR phase, as shown in Figure 4.2, compares the generic requirements of the clusters with the descriptor sets of the transactions within the same clusters. For each transaction, the modification efforts from the minimum requirement and from the maximum requirement of the same cluster are evaluated. The framework with the smaller value is recommended. Adaptation of the recommended frameworks to the implementation of individual transactions is based on the comparison between the descriptor sets of the transactions and that of the frameworks. Sections 4.3 and 4.4

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

prescribe steps for the implementation of individual transactions through the reuse of the frameworks.

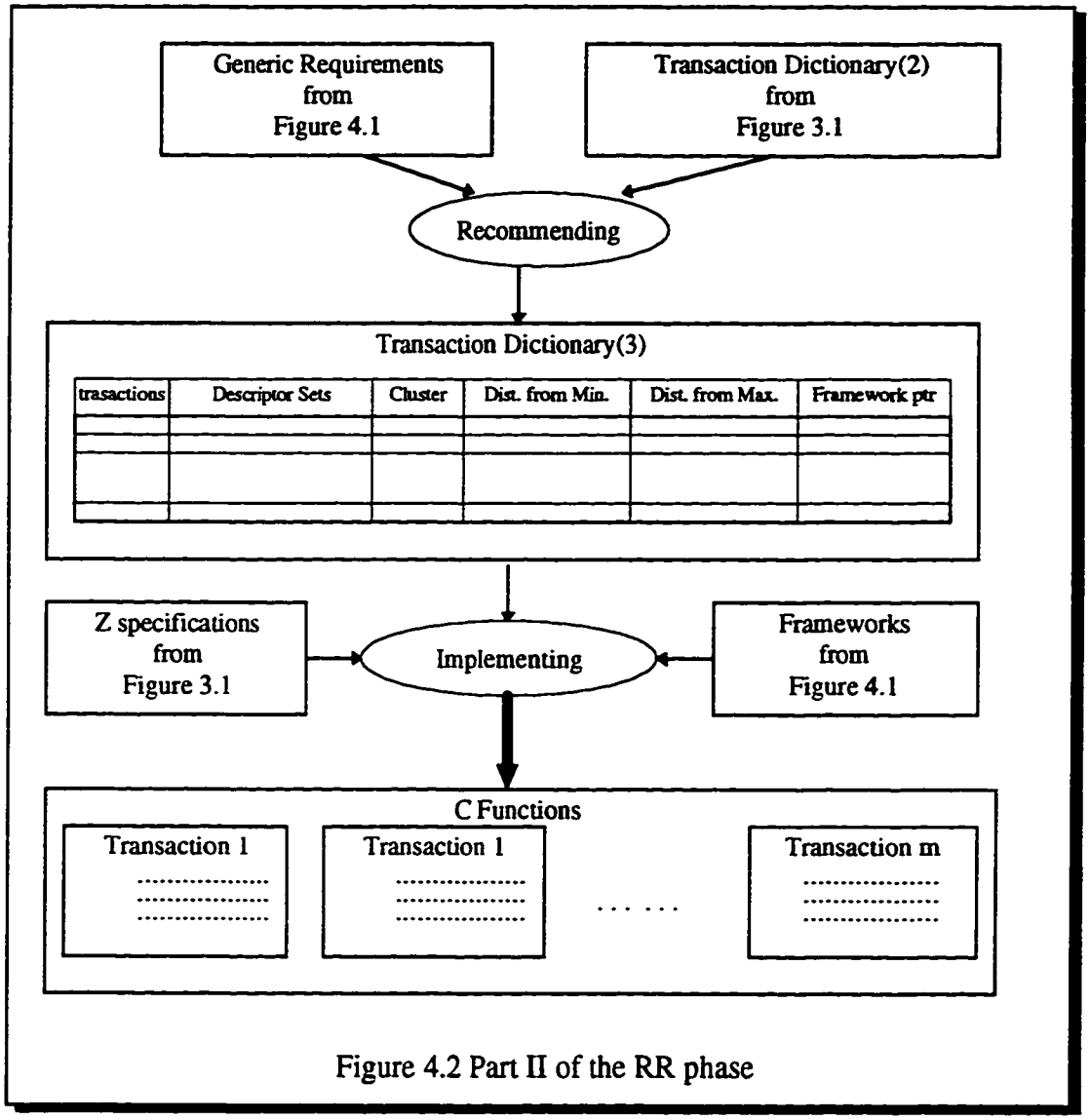

Figure 4.2 Part II of the RR phase

4.1 Extracting Generic Requirements

The repository of Z schemas constitutes the starting point for defining reusable frameworks. It is appropriate and natural to analyze transactions that are in the same cluster to identify the "key" operations for each cluster. From the transactions grouped

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

within clusters, we extract generic requirements of functionality that are exploited by multiple transactions. Three perspectives of such knowledge are of value: the *minimum*, *maximum*, and *majority* requirements. Sections 4.1.1, 4.1.2 and 4.1.3 define algorithms for extracting these requirements, respectively.

4.1.1 Minimum requirements

The minimum requirements for a cluster of transactions is defined as the set of basic operations shared by every transaction in the cluster along with the associated minimum level of effort. Such information is extracted from the descriptor sets that are used to describe the transactions. If t_1 and t_2 are two transactions and d_1 and d_2 are the corresponding descriptor sets, the minimum requirement of the two transactions, denoted by $\min(d_1, d_2)$ is also a descriptor set as determined by the algorithm in Figure 4.3. The algorithm searches for the minimum functionality required by both transactions. When an operation is only requested by one of the transactions, it does not contribute to the minimum requirements. When an operation is requested by both transactions, we include the operation in the minimum requirements but need to look further to determine the appropriate value. If the associated values for the operation are the same for both transactions, the appropriate value for the operation in the minimum requirement descriptor set is the value shared by the two transactions. If not, the determination of the appropriate value is affected by the type of the basic operation. For the cases of *select* and *join* operations, the values indicate the number of relations involved in the operations. Therefore, the smaller value of the two transactions reflects the minimum number of relations that needs to be considered for the minimum

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

requirement of the two transactions. For the cases of *insert*, *delete*, and *update* operations, the associated values for the operations represent the number of intra and/or inter relation constraints that the operations need to preserve during the execution of the operations. Thus, we use the smaller value of the two transactions as the minimum number of constraints that need to be considered by both transactions. For the *input* operation, the interface required for getting a value input is much easier than that for a set of values, so "value" is associated with the "input" operation when two transactions request different types of input interface. Similarly, <output, old> is chosen when two different values are used in two transactions.

$\min(d_1, d_2) = \varnothing$;
For each $op_{ik} = op_{jl}$ do
  If (op_{ik} = "input")
    If ($v_{ik} = v_{jl}$)
      then $\min(d_1, d_2) = \min(d_1, d_2) \cup \{< op_{ik}, v_{ik}>\}$
      else $\min(d_1, d_2) = \min(d_1, d_2) \cup \{< op_{ik}$, value$>\}$
  If (op_{ik} = "output")
    If ($v_{ik} = v_{jl}$)
      then $\min(d_1, d_2) = \min(d_1, d_2) \cup \{< op_{ik}, v_{ik}>\}$
      else $\min(d_1, d_2) = \min(d_1, d_2) \cup \{< op_{ik}$, old$>\}$
  If op_{ik} is in {"select", "join", "delete", "insert"}
    then $\min(d_1, d_2) = \min(d_1, d_2) \cup \{< op_{ik}, \min\{ v_{ik} , v_{jl} \}>\}$;
  If op_{ik} = "update"
    then $\min(d_1, d_2) = \min(d_1, d_2) \cup \{<op_{ik},(\min\{v_{ik1},v_{jl1}\}, \min\{v_{ik2},v_{jl2}\})>\}$;

Figure 4.3 Minimum requirements of two transactions

As a result, for any two transactions t_1 and t_2 with the descriptor sets d_1 and d_2, respectively, $\min(d_1, d_2)$ returns a descriptor set representing the minimum requirement of the two transactions. Let $t_1, t_2, \ldots, t_n$ be transactions belonging to cluster C and $d_1, d_2, \ldots, d_n$ be the descriptor sets corresponding to the transactions, respectively. Since

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

the minimum requirement function of two transactions, i.e. $\min(d_i, d_j)$, returns a descriptor set, the minimum requirements of the cluster C, MIN(C), is determined recursively by $\min(d_i, d_j)$. The algorithm and a graphical depiction of the process are illustrated in Figure 4.4.

If C contains only one transaction t_1
then $MIN(C) = d_1$;
else
partition C into C_1 and C_2
$MIN(C) = \min(MIN(C_1), MIN(C_2))$

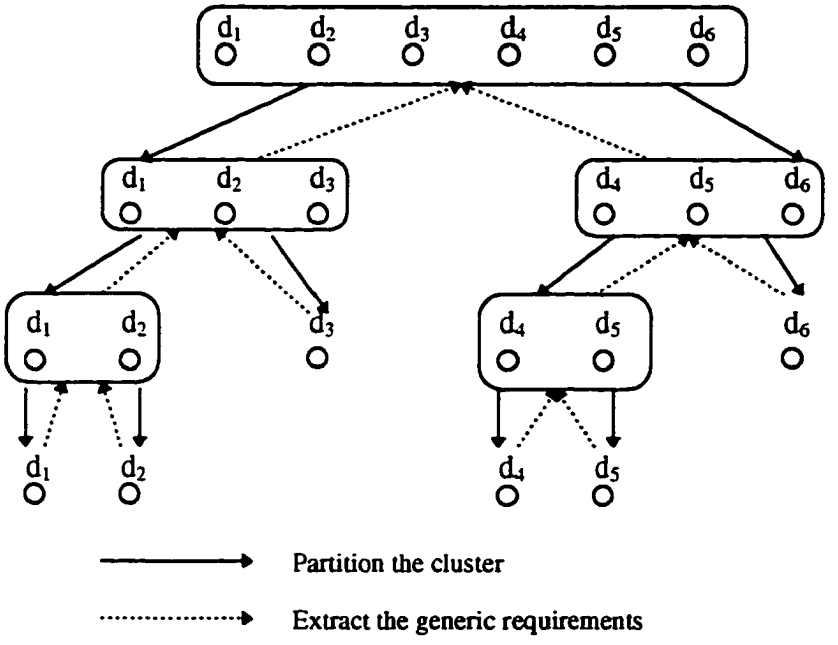

Figure 4.4 Minimum requirements of a cluster of transactions

4.1.2 Maximum requirements

The maximum requirement of a cluster of transactions includes basic operations requested by transactions in the cluster with the associated maximum level of effort. Such information is extracted from descriptor sets that are used to describe the transactions. If t_1 and t_2 are two transactions and d_1 and d_2 are the corresponding

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

descriptor sets, the maximum requirement of the two transactions, denoted by $max(d_1,d_2)$ is also a descriptor set determined from the algorithm in Figure 4.5.

```
max(d1, d2) = ∅;
For each opik = opjl do
   If (opik = "input")
      If (vik = vjl)
         then max(d1, d2) = max(d1, d2) ∪ {< opik, vik>}
         else max(d1, d2) = max(d1, d2) ∪ {< opik, set>}
   If (opik = "output")
      If (vik = vjl)
         then max(d1, d2) = max(d1, d2) ∪ {< opik, vik>}
         else max(d1, d2) = max(d1, d2) ∪ {< opik, new>}
   If opik is in {"select", "join", "delete", "insert"}
      then max(d1, d2) = max(d1, d2) ∪ {< opik, max{ vik , vjl }>};
   If opik = "update"
      then max(d1, d2)=max(d1, d2) ∪{<opik,(max{vik1,vjl1}, max{vik2,vjl2 })>};
For each opik that does not have matching opjl do
   If opik = "update"
      then max(d1, d2)=max(d1, d2) ∪ {<opik, (vik1, vik2)>}
      else max(d1, d2)=max(d1, d2) ∪ {<opik, vik >};
For each opjl that does not have matching opik do
   If opjl = "update"
      then max(d1, d2)=max(d1, d2) ∪ {<opjl, (vjl1, vjl2)>}
      else max(d1, d2)=max(d1, d2) ∪ {<opjl, vjl >};
```

Figure 4.5 Maximum requirements of two transactions

We are searching for the maximum functionality requested by both transactions. When a <descriptor, value> pair contains an operation that is only requested by one of the two transactions, the pair becomes a member of the maximum requirement descriptor set. When an operation is requested by both transactions, we include the operation in the maximum requirement but need to look further to determine the appropriate value. If the associated values for the operation are the same for both transactions, the appropriate value for operation in the maximum requirement

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

descriptor set is the value shared by two transaction. If not, the determination of the appropriate value is affected by the type of the operation. For the cases of *select* and *join* operations, the values indicate the number of relations involved in the operations required by each transaction. Therefore, the larger value of the two transactions reflects the maximum number of relations that needs to be considered for the maximum requirement of the two transactions. For the cases of *insert*, *delete*, and *update* operations, the associated values for the operations represent the number of intra and/or inter relation constraints that the operations need to preserve during the execution of the operations. Thus, we use the larger value of the two transactions as the maximum number of constraints that need to be considered by both transactions. For the *input* operation, the interface required for getting a value input is much easier than that for a set of values, so "set" will be associated to the "input" operation when two transactions request different types of input interface. Similarly, <output, new> will be chosen when two different values are associated with the output operations in the two transactions.

As a result, for any two transactions t_1 and t_2 with d_1 and d_2 as descriptor sets, respectively, $\max(d_1, d_2)$, as described in Figure 4.5, returns a descriptor set representing the maximum requirement of the two transactions. Let $t_1, t_2, \ldots, t_n$ be transactions belonging to cluster C and $d_1, d_2, \ldots, d_n$ be the descriptor sets corresponding to the transactions, respectively. The maximum requirements of the cluster, MAX(C), is also a descriptor set which is determined recursively, as shown in

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

Figure 4.6, by the maximum requirement function of two transactions, $\max(d_i, d_j)$ where $1 \leq i, j \leq n$.

```
If C contains only one transaction t_i
   then MAX(C) = d_i;
   else
        partition C into C_1 and C_2
        MAX(C) = max(MAX(C_1), MAX(C_2))
```

Figure 4.6 Maximum requirements of a cluster of transactions

4.1.3 Majority requirements

The majority requirement of a cluster of transactions is defined as a descriptor set that contains the <descriptor, value> pairs that appear in the majority transactions of the cluster. Only those <descriptor, value> pairs that are requested by at least 50 percent of all transactions are included in the final descriptor set. Let $t_1, t_2, \ldots, t_n$ be transactions belonging to cluster C and $d_1, d_2, \ldots, d_n$ be the descriptor sets corresponding to the transactions, respectively. Figure 4.7 defines the algorithm for finding the majority requirement of the cluster. To extract such information, we need to first determine the frequency of each pair of <descriptor, value> in the cluster. For those descriptor sets that do not include all the basic operations, dummy pairs of <descriptor, value> are inserted. We use NULL as the value in the dummy pairs to indicate the corresponding operation is not requested by the transaction. The frequencies are divided by the total number of transactions in the cluster. After the descriptor set of the majority requirements is determined, dummy pairs are deleted from the final descriptor set.

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

```
For all d_i, 1 ≤ i ≤ n do
   For each basic operation op in {"input", "output", "select", "join",
                                   "insert", "delete", "update"} do
      If (op is not a descriptor in d_i) Then
         extend d_i with <op, null>
For each distinct <op, val> pair, determine the number of occurences
and denoted as (<op, val>, freq).
totfreq = 0;
For all (<op, val>, freq)do
   totfreq = totfreq + freq
Maj_req = { }
For all (<op, val>, freq)do
   If ((val ≠ NULL) && ((freq / totfreq) > 0.5))
      Maj_req = Maj_req ∪ {<op, val>}
```

Figure 4.7 Majority requirement of a cluster of transactions

4.2 Generation of Frameworks

By analyzing the transactions within a cluster, generic requirements of functionality are extracted. Section 4.1 described three different perspectives for such generic requirements. In addition, algorithms were given to facilitate automatic extraction of the generic requirements. There are several benefits of designing frameworks to accomplish generic requirements. First, these requirements take the same representation scheme as that used for individual transactions, i.e. descriptor sets. The purpose of extracting such knowledge is to enable the generation of frameworks for each cluster so that implementation of individual transactions in the same cluster can reuse the frameworks. The descriptor sets measure the basic operations along with the level of effort required by the frameworks. By designing a single framework to serve a number of distinct transactions, the cost of developing the framework can be amortized over those transactions. Second, the implementation of different transactions

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

from the same framework can lead to a desirable uniformity of the system. Therefore, the system will be easier to understand and hence maintain. Third, the fact that the framework will be a template for several transactions forces the developers of the framework to strive for a high quality product which in turn can lead to applications of higher quality. Fourth, since reusing an existing framework is easier than implementing a transaction, in many cases designing frameworks can be allotted to a small team of highly skilled engineers.

As a result of the reusability analysis phase of RRM, as described in Chapter 3, transactions grouped into the same cluster share several features. Section 4.1 described the search for such shared features by presenting algorithms that automatically extract generic requirements for each cluster. This section provides a systematic way to generate design frameworks to accomplish the generic requirements described in section 4.1. Such a process ensures that the generated frameworks are *useful* for the implementation of individual transactions belonging to the same cluster as the frameworks. In fact, the frameworks serve as templates for the implementation of individual transactions in the same cluster. In this section, a systematic way is defined to generate frameworks that are *reusable* by individual transactions.

In order to build a framework that is reusable, environment information such as design decisions made toward the implementation and guidelines for modification need to be provided for the reuser. Section 4.2.1 lists the design decisions. The reusable framework specifies an implementation of the generic requirement based on the design

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

decisions. Section 4.2.2 describes a systematic way to automatically generate a reusable framework given the descriptor set of a generic requirement.

4.2.1 Design decisions

The descriptor sets of the generic requirements only describe "what" the frameworks intend to do. To specify "how" they are done, design decisions need to be made before the actual implementation of the requirements. Such knowledge includes choice of language and support provided by the language. More detailed information such as the data structure to use and performance requirements can also influence the generation of the framework. In this research, we use the high level programming language C to implement the transactions of the database applications. Therefore intra or inter relation constraints need to be explicitly checked by the program before any update upon the contents of the database is committed. Each tuple of a relation is defined by a *struct* type and the relation itself is defined as an array of struct. For each relation, an additional variable needs to be declared to store the size of the relation. Transactions operate on the contents of the database through *functions*. However, the reusable frameworks are not executable C functions but are structurally similar to the format of a C function. The reason for doing so is that the descriptor set only represents the functional semantics information of the framework. For example, if <join, 3> is a member of the descriptor set, we know that a *join* operation is to be performed on three relations. What we do not know are the actual names of the relations and the matching criteria among the relations. On the other hand, if <output, old> is a member of the descriptor set, we know that an intermediate relation is

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

necessary to store the query result. In additon, the intermediate relation takes the structure of an existing relation even though the name of this existing relation is unknown. Adaptations, such as substitution and/or specialization, need be made during the implementation of individual transactions to fulfill the conceptual meaning of the transactions. The reusable framework takes into account all basic operations described in the generic requirements.

Additional useful knowledge includes the built-in functions that are available to transactions and information on how to access those functions.

4.2.2 Reusable framework

The structure of a reusable framework, as shown in Figure 4.8, consists of three sections:

1. The interface of the framework defines how the framework communicates with its environment:
   - Name of the framework: Cluster#_type, where *type* indicates whether the framework accomplishes the minimum, maximum, or majority requirements.
   - Input parameters: relations that contain information necessary for the completion of the functionality requested by the descriptor set.
   - Output parameters: relations that will be updated by the execution of the framework.
2. The local variables declares intermediate relations, index variables, and variables to store input information. These variables are private to the framework and are not accessible outside the framework.

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

3. The body of the framework defines actions upon the contents of the database actually.

Interface:
Name
Input Parameters
Output Parameters
Local Variables:
Index Variables
Input Media
Struct Definition
Resulting Relation
Body:

Figure 4.8 The structure of frameworks

First, a meaningful name is assigned to each framework. The name of a given framework is determined based on two things: the cluster to which the framework belongs and the type of requirement that the framework accomplishes. For instance, if a framework is designed to accomplish the minimum requirement of Cluster 2, the name of the framework will be "Cluster2_Min". On the other hand, a framework that accomplishes the maximum requirement of the fifth cluster will have "Cluter5_Max" as its name.

The interface of the framework also defines the input and output parameters of the function. The input parameters are those relations that participate in read-only and/or update operations of the framework. No change will be made to the contents of these relations by the framework. Let $d = \{<op_1, val_1>, <op_2, val_2>, ..., <op_m, val_m>\}$ be the descriptor set of a given generic requirement r, then Figure 4.9 defines an algorithm to assist the determination of the input parameters of the framework for r.

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

```
For all <op, val> ∈ d do
   If (((op = = "select") or (op = = "join")) and (val > 0))
        print op
        for (i = 0; i < val; i++)
            print "relation_" & i, "size_relation_" & i
        change line
   If ((op = = "insert") and (val > 0))
        print "insert"
        for (i = 0; i < val; i++)
            print "pk_relation_" & i, "size_relation_" & i
        change line
    If ((op = = "delete") and (val > 0))
        print "delete"
        for (i = 0; i < val; i++)
            print "fk_relation_" & i, "size_relation_" & i
        change line
    If ((op = = "update") and ((val1 + val2)> 0)))
        print "update"
        if (val1 > 0)
            for (i = 0; i < val1; i++)
                print "pk_relation_" & i, "size_relation_" & i
        if (val2 > 0)
            for (i = 0; i < val2; i++)
                print "fk_relation_" & i, "size_relation_" & i
        change line
```

Figure 4.9 Determination of input parameters for framework

When a *select* operation is requested by the generic requirement, e.g. <select, 3> is in the descriptor set, the function needs the relations involved in the operation in order to make a selection of tuples that satisfy certain criteria. Therefore, three relations need to be included as input parameters. In addition, the sizes of the relations need to be passed as input parameters due to the design decisions that arrays of struct are used to represent relations. The same analogy applies to *join* operation. For the cases of *insert*, *delete*, and *update* operations, the associated values indicate the number of inter relations that the function needs to check before actual changes to the data can be

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

made. Therefore, those relations and their corresponding sizes need to be passed as input parameters too.

The output parameters are those relations in the database whose data may be changed by the function. Let d = {<op_j, val_j>} be the descriptor set of a given generic requirement r, then Figure 4.10 defines an algorithm to assist the determination of the output parameters of the framework for r. When an *insert*, *delete* or *update* operation is requested by the framework, the relations to be affected by the operation are the output parameters of the framework. As a result, the interface of a framework consists of information such as the total number of relations involved in each basic operation and the corresponding size of the relations. Keep in mind that some relations may, and are very likely to, participate in more than one operation.

```
If there exists no pair of <insert, ?> , <delete, ?> or <update, ?> in d
   print "None"
   terminate
For all  <op, val> ∈ d do
   If  (op = = "insert")
        print "insert", "ins_relation", "size_relation"
        change line
   If  (op = = "delete")
        print "delete", "del_relation", "size_relation"
        change line
   If  (op = = "update")
        print "update", "upd_relation", "size_relation"
        change line
```

Figure 4.10 Determination of output parameters for framework

After determining the interface of the function, private (local) variables need to be defined. The local variables are accessible only inside the framework. They are used to store information that is useful for the completion of the framework but not

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

necessary to be kept as contents of the database. Let d = {<op_j, val_j>} be the descriptor set of a given generic requirement r, then Figure 4.11 defines an algorithm to assist the determination of the local variables of the framework for r.

```
Max_ind = -1
For all <op, val> ∈ d do
   If (val > max_ind) and (op ∉ {"input", "output"})
      Max_ind = val
If (Max_ind > 0)
  print "Index Variables:"
  for (i = 0; i < Max_ind; i++)  print "Ind_" & i
  change line
If (<input, value> ∈ d) Then
  print "Input media: datatype value;"
If (<input, set> ∈ d) Then
  print "Input media: struct in_set { [datatype value]+ };
If (<output, old>∈ d) Then
  print "Resulting relation: struct existing_relation result[]"
  print "int size_result;"
If (<output, new> ∈ d) Then
  print "struct definition: struct relation_new {[domain field;]+};"
  print "Resulting relation: struct relation_new result[];"
  print "int size_result;"
```

Figure 4.11 Determination of local variables for framework

When <input, value> belongs to the descriptor set, a single variable needs to be declared to store the input information. When <input, set> belongs to the descriptor set, an array needs to be defined to store the input information. An array of struct needs to be declared if *output* is in the descriptor set. Depending on the value associated with it, detail type definition may be necessary. In addition, index variables are necessary in order to search through contents of the relations involved. At this moment, no actual data types of the variables declared are determined. The framework only sets up a template to be reused by the implementation of individual transactions.

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

The body of the framework needs to be set up to coordinate basic operations requested by the descriptor set. There are four main sections in the body: input, retrieval, update, and output sections. Let $d = \{<op_j, val_j>\}$ be the descriptor set of a given generic requirement r, Figure 4.12 provides guidelines to assist the determination of the body of the framework for r.

The first section is the input section. Getting necessary input information can be done before the other operations. It does not matter to the transaction *how* the input information is received, therefore, the template only decides what built-in functions: *get_value* or *get_set* should be called. The input section is followed by the retrieval section that retrieves necessary information into the intermediate relation. The value associated with *output* operation in the descriptor set indicates that an intermediate relation is necessary to store the data retrieved by read queries. Depending on the structure of the resulting relation, different approaches are used to collect tuples for the relation. The *select* operation searches through relation(s) independently and filters those tuples that satisfy certain criteria. FOR loops are used to facilitate the search. Searches need to be performed for *join* operations too. The difference is that now searching is performed by looking at all relations involved in the operation simultaneously. A nested FOR loop is required. When both *select* and *join* are requested by the descriptor set, it is more efficient to perform the join operation on the relations after the select operation. To do so, the template uses the nested FOR loop required by *join* operations and searches for the relations that are also involved in the *select* operations. The third section is the update section where the contents of the

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

```
/* the input section */
If (<input, value> ∈ d) Then
   call built-in function Get_value(&value)
If (<input, set> ∈ d) Then
   call built-in function Get_set(&in_set)
/* the retrieval section */
If (<join, m> ∈ d) Then
   { set up a nested FOR loop with m layers
   IF (<select, n> ∈ d) Then
        Add select condition to join condition
   }
Else If (<select, n> ∈ d) Then
        set up n separate FOR loops
/* the update section */
If (<insert, m> ∈ d) Then
   set up m separate FOR loops to check on the primary key constraints
   If no constraint is violated Then
       call the built-in function Add_rec to insert record
   Else Display warning msgs
If (<delete, m> ∈ d) Then
   set up m separate FOR loops to check on the foreign key constraints
   If no constraint is violated Then
       call the built-in function Delete_rec to delete record
   Else Display warning msgs
If (<update, (m, n)> ∈ d) Then
   set up m separate FOR loops to check on the primary key constraints
   set up n separate FOR loops to check on the foreign key constraints
   If no constraint is violated Then
       call the built-in function Add_rec to insert record
   Else Display warning msgs
/* the output section */
Call built-in function Dispaly(result, size_result)
```

Figure 4.12 Determination of the body for framework

relations in the database may be changed. The *insert*, *delete*, and *update* operations need to be performed after the *select* and *join* operations have been completed. When the associated values for the three operations are zero, i.e., no inter relation constraints

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

may be violated by the operations, one FOR loop is sufficient to search through the output relation in order to modify the contents of the database. Otherwise, like the case of *join*, the nested FOR loop is necessary to make sure the constraints are not violated by the operations. The last section is the output section. Displaying the resulting relation is always done after the other operations by calling the built-in function *print_relation*.

4.3 Implementation of existing transactions

A systematic way is needed to extract the generic requirements and to generate reusable frameworks for each cluster of transactions. Three kinds of generic requirements were defined: the minimum, maximum, and majority requirements. For any transaction in a selected cluster, there are three frameworks available to assist the implementation of the transaction. The selection of the best candidate is discussed in Section 4.3.1. The implementation of the transaction is a process of adapting the selected framework. Section 4.3.2 presents the adaptation process.

4.3.1 Select framework

Since RRM is intended to capture the view of reuse that a component can be constructed by modifying another component, we are looking for the framework that requires the least amount of modification for the implementation of any given transaction. The frameworks generated for clusters accomplish the generic requirements for the transactions of the same cluster. Therefore the framework to be reused by the implementation of a given transaction is chosen from the three reusable

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

frameworks of the same cluster as the transaction. Let $d_t = \{< op_{t1}, v_{t1}>, < op_{t2}, v_{t2}>, . . ., < op_{tm}, v_{tm}>\}$ and $d_f = \{< op_{f1}, v_{f1}>, < op_{f2}, v_{f2}>, . . ., < op_{fn}, v_{fn}>\}$ be the descriptor sets for a given transaction and framework, respectively. The degree of effort required to go from the framework to the transaction is estimated according to the algorithm shown in Figure 4.13.

When the descriptor set for the transaction is the same as that for the framework, the modification effort is the minimum. The only adaptation necessary is substituting the right relations of the transaction to those in the reusable template of the framework. We define this as the zero point of the effort. For *input* and *output* operations that are requested by both descriptor sets, extra effort is necessary when one operation tries to read in a value while the other operation tries to read in a set; or when one results in a relation that takes the structure of an existing relation and the other requires a new structure for its resulted relation. Extra effort is also needed when the number of relations or the number of intra and/or inter relation constraints are different between two descriptor set for the case of *select*, *join*, *insert*, *delete*, and *update*. The increasing amount of effort depends on the difference of the two corresponding values. Even though some programmers may prefer removing extra operations from the framework over adding necessary operations to the framework and vice versa, we treat the two situations evenly. When *input* and *output* operations are only requested by one of the two descriptor sets, the modification effort depends on the value associated with the operation that is requested. The effort in adding functionality to get a set of input information is more than that of getting a single input value. Using

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

effort = 0;
For each $op_{ti} = op_{fj}$ do
  If (op_{ti} = "input" or op_{ti} = "output") and ($v_{ti} \neq v_{fj}$)
    then effort = effort + E_{op};
  If (op_{tk} = select" or op_{tk} = "join")
    then effort = effort + abs(v_{ti} - v_{fj}) * E_{op};
  If (op_{ti} = "insert")
    then effort = effort + abs(v_{ti} - v_{fj}) * E_{pk};
  If (op_{ti} = "delete")
    then effort = effort + abs(v_{ti} - v_{fj}) * E_{fk};
  If op_{tk} = "update"
    then effort =effort + abs(v_{ti1} - v_{fj1}) * E_{pk} + abs(v_{ti2} , v_{fj2}) * E_{fk};
For each op_{ti} that does not have matching op_{fj} do
  If (op_{ti} = "input" and v_{ti} = "value") or (op_{ti} = "output" and v_{ti} = "old") then
    effort = effort + E_{op};
  If (op_{ti} = "input" and v_{ti} = "set") or (op_{ti} = "output" and v_{ti} = "new") then
    effort = effort + 2 * E_{op};
  If (op_{ti} = "select" or op_{ti} = "join")
    then effort = effort + v_{ti} * E_{op};
  If (op_{ti} = "insert")
    then effort = effort + v_{ti} * E_{pk} + E_{ins};
  If (op_{ti} = "delete")
    then effort = effort + v_{ti} * E_{fk} + E_{del};
  If op_{ti} = "update"
    then effort = effort + v_{ti1} * E_{pk} + v_{ti2} * E_{fk} + E_{del};
For each op_{fj} that does not have matching op_{ti} do
  If (op_{fj} = "input" and v_{fj} = "value") or (op_{fj} = "output" and v_{fj} = "old") then
    effort = effort + E_{op};
  If (op_{fj} = "input" and v_{fj} = "set") or (op_{fj} = "output" and v_{fj} = "new") then
    effort = effort + 2 * E_{op};
  If (op_{fj} = "select" or op_{fj} = "join")
    then effort = effort + v_{fj} * E_{op};
  If (op_{fj} = "insert")
    then effort = effort + v_{fj} * E_{pk} + E_{ins};
  If (op_{fj} = "delete")
    then effort = effort + v_{fj} * E_{fk} + E_{del};
  If op_{fj} = "update"
    then effort = effort + v_{fj1} * E_{pk} + v_{fj2} * E_{fk} + E_{upd};

Figure 4.13 Modification effort

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

the structure of an existing relation is considered easier than defining a new structure to store resulting information. The same consideration applies to the operations *select*, *join*, *insert*, *delete*, and *update*. The values associated with the operations, i.e. the number of relations or the number of constraints, reflect the effort necessary for modification.

At the end of the reusability analysis phase of RRM, a cluster structure is determined for the transactions that are identified in the specification of the application. Each transaction is measured by a descriptor set and belongs to exactly one cluster. For each cluster, the algorithms presented in Section 4.1 derive three descriptor sets each corresponding to one reusable framework for the cluster. For each transaction, the degree of effort required to modify each framework (minimum, maximum, and majority) of the same cluster to implement the transaction is calculated. The framework that is recommended for reuse by the given transaction is the one that has the minimum value.

4.3.2 Adaptation process

The Z schemas of the transactions provide information to assist the modification of the frameworks. The transactions are implemented by C functions. When a transaction and the selected framework share the same descriptor set, the adaptation process is simple. First the name of the framework is replaced by the name of the schema. Next, the input and output parameters of the reusable framework contribute to the prototype of the function. The prototype consists of a list of relations and their corresponding sizes. Notice that for the input and output parameters of the

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

framework, we identify the number of relations involved in different type of operations. When implementing the transaction, the names of the relations involved are available. If one relation participates in two operations, it occurs only one time in the prototype. The local variables of the framework are used to store intermediate information. There are no actual data types associated with them. A visit to the data dictionary of the database is necessary to fill in such detail about the transaction. Because the transaction and the framework share the same descriptor set, the overall structure of the body of the framework remains unchanged for the transaction. Only substitutions need to be made to replace the corresponding actual relations and conditions.

The difference between the descriptor set for the transaction and that of the framework indicates further modification of the framework. When the selected framework is the minimum framework of the cluster, extra operations need to be added to the framework to generate implementation of the transaction. When the selected framework is the maximum framework, redundant operations need to be removed from the framework. When the majority framework is selected, both adding and removing may take place. Implementation of the functional prototype is similar to the situation of an equivalent descriptor set between the transaction and the framework. Modification to the local variable declaration section is not limited to associating each variable with the actual data type. When there exist differences in the *input* and *output* operations between the framework and the transaction, new variables need to be declared. The difference in the *input* and *output* operations also affects the implementation of the body of the transaction. If *input* is requested by the framework but not the transaction,

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

the variable(s) declared for storing input information need to be removed. The call to the built-in function *Get_value* or *Get_set* also needs to be removed from the framework. If the opposite is true, an additional variable(s) needs to be declared to store input information for the transaction. And a call to either *Get_value* or *Get_set* needs to be made depending on what type of input information is requested. When *input* is requested by both the framework and the transaction but the values associated with it are different, modification to the variable declaration is necessary. And a different function call needs to be made. The same analogy applies to the intermediate relation that corresponds to the *output* operation. When the difference between the descriptor sets of the framework and the transaction involves other operations, a FOR loop may be added to or removed from the body.

4.4 New Transaction

Section 4.3 discussed the process of implementing the transactions that are specified at the specification phase, i.e. before performing the reusability analysis. After the system is submitted to the clients, very often new functionality will be desired as time goes on. RRM provides a cost-effective way to allow the design of a new transaction and can easily retrieve the best candidate reusable component and adapt it.

4.4.1 Specify the transaction

When a new transaction is requested, it is formally specified by a Z schema according to guidelines provided in Section 3.1. The Z schema indicates *what* the transaction intends to accomplish. No detailed information of *how* such requirement

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

can be done is involved. The schema then is sent to the REST system which parses the schema to generate a descriptor set indicating what basic operations are requested by the transaction and the associated level of effort. Now the transaction is represented under the same measurement space as the reusable frameworks in the repository. The next step is to select the most appropriate framework to assist the implementation of the transaction.

4.4.2 Retrieval of candidate framework

Instead of searching through all of the existing, implemented transactions in the application, the repository of frameworks is searched. The size of the repository is much smaller than the number of the transactions in the application since each framework is designed to accomplish a generic requirement shared by a cluster of transactions in the application. There are five attributes associated with each framework: the name of the framework, the cluster it belongs to, the type of the generic requirement it accomplishes, the descriptor set, and the associated design. By applying the modification effort algorithm, as shown in Figure 4.5 of Section 4.3, the effort needed to adapt each framework to the implementation of the desired transaction is estimated. The framework that provides the minimum value is chosen as the candidate to be reused by the desired transaction. The transaction is classified with the cluster to which the selected framework belongs.

4.4.3 Implementation

The candidate framework serves as an implementation template for the new transaction. Since the transaction is represented under the same measurement space as

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

transactions previously defined, the descriptor set of the transaction can be compared to the descriptor set of the selected framework to assist the adaptation. In fact, the implementation of the new transaction can follow the same adaptation process as that for existing transactions as described in Section 4.3.2.

4.5 Summary

This chapter described the reusability realization phase of RRM. A systematic and pragmatic approach for extracting generic requirements for a selected cluster of transactions was presented in Section 4.1. The generic requirements were fulfilled by reusable frameworks. This approach makes sure that the frameworks generated are useful for implementation of transactions in the same clusters. The design decisions and methods for generating such frameworks were also provided in Section 4.2 to make sure that the frameworks are also usable by the implementation of the transactions in the same cluster.

The implementation for an individual transaction reuses the frameworks that are functionally most similar to the transaction. For existing transactions, i.e. the transactions that are identified and specified before the reusability analysis phase of RRM, the selection of the framework is performed by comparing the difference between the transaction to be implemented and the three reusable frameworks for the same cluster to which the transaction was assigned in the reusability analysis phase. The modification effort is evaluated based on the algorithm in Figure 4.5. For a new transaction, i.e. a new functionality requested after the delivery of the application, there

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

is no knowledge of which cluster the desired transaction may belong to. However, the retrieval is not performed against all transactions in the application but rather against only the repository of the frameworks derived for the application. The retrieval of the framework is performed by evaluating the modification effort from each framework against the desired transaction. The framework that provides the smallest value is selected as the candidate reusable framework. The adaptation process, provided in Section 4.3.2, is the same for both existing transactions and new transactions.

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

Chapter 5

Case Study

In this research, we defined a requirement reusability model (RRM) that utilizes a formal specification language and statistical techniques to detect reusability potential of software components. The two major phases of RRM are the reusability analysis phase and the reusability realization phase. A system, called REST (REuser's assiSTant), implements the model. The assumption of RRM is that a relational database application is formally specified according to plans and rules of discourse (Section 3.1.1). In this chapter, a case study is presented to demonstrate the capabilities of RRM. The data structure for the case study problem was specified in Z schemas in Section 1.3.3. The data dictionary that accompanies these schemas is given in Section 5.1. The database system allows its users to manipulate the contents of the database through the transactions listed in Table 5.1.

The transactions are specified using Z schemas in section 5.2. Section 5.3 follows the steps of the RA phase of RRM and generates a cluster structure. Section 5.4 follows the steps of the RR phase of RRM and generates reusable frameworks for the clusters defined in Section 5.3. The implementation of the transactions is also presented in Section 5.4.

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

Table 5.1 List of transactions

1	Get full details of all finished projects
2	Get full details of all employees who earn minimum wage
3	Get full details of all project assignments of male employees
4	Get Sex of all employees worked on project p1
5	Get full detail of all male employees
6	Get IDs of all female managers of department in New York
7	Get IDs and Rates of employees who work more than 100 hours on project p2
8	Get IDs of employees who work on the project controlled by the same department that hires them
9	Get all departments in New York whose managers' basic rates are higher than 50
10	Get all departments whose managers are male
11	Get all projects in New York that are controlled by departments whose managers are female
12	Get ID and Sex of employees whose basic rates are under 10
13	Get project numbers and hours that a given employee has worked on
14	Get IDs of departments that are in a given city
15	Get ID and Rate of employees who are hired after a given project started and work on the project
16	Get all details of the projects that are in the same city as the departments control them
17	Get all details of the projects that controlled by a given department before its new manager was hired
18	Get full details of departments in New York
19	Get full details of all projects controlled by a given department
20	Get ID and Rate of female employees who work on projects in New York
21	Get all employees who are hired between two given dates
22	Get employee IDs who work less than 10 hours on a given project
23	Get department ID and its manager's basic rate for the departments that control at least one projects in New York
24	Get IDs and hours of the project that worked by female employees
25	Get ID and basic rate for male employees.

5.1 Data Dictionary

According to Date[Dat 90], a data dictionary contains information concerning objects that are of interest to the database system. Examples of such objects are base tables, views, indices, and cross-reference information. The contents of the dictionary

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

can be regarded as "data about the data". Such information is essential for the system to function properly. This research focuses on transactions applied to the data stored in the database system. Therefore, the objects of interest include information such as the structure of the base tables and integrity rules. The data dictionary itself can be regarded as a relation where each tuple of the relation keeps track of constraints applied to one attribute in the database system. There are five fields of interest for each attribute in the database: the name of the attribute, the domain from which the attribute can draw values, the relation to which the attribute belongs, the intra relation constraints to which the values of the attribute should obey, and the inter relation constraints that may be affected by the changes to the value of the attribute.

The resulting data dictionary for the case study problem is displayed in Table 5.2. According to Figure 1.5, there are four relations in the database system: EMPL, DEPT, PROJ, and WORK. The figure also reveals the name of the attributes in each relation along with their corresponding domains. The names of the attributes, the domains from which the attributes draw their values and the relations to which the attributes belong are the first three columns in Table 5.2. There were five schemas specified in Section 1.3.3. The first four schemas specified the intra relation constraints in their predicate sections. The fourth column lists the intra relation constraints applicable to the attributes. The database schema DB indicates the inter relation constraints that the database has to preserve at any given moment. The inter relation constraints applicable to the attributes are found in the fifth column.

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

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

Table 5.2 Data dictionary

Name	Domain	Relation	Intra relation constraints	Inter relation constraints
ENum	ENUM	EMPL	$\forall e : empls \bullet e.ENum \neq Null \wedge$ $\forall e_1, e_2 : empls \bullet e_1.ENum = e_2.ENum \Leftrightarrow e_1 = e_2$	$\forall d : depts \bullet (\exists e : empls \bullet e.ENum = d.ManENum) \wedge$ $\forall w : works \bullet (\exists e : empls \bullet e.ENum = w.ENum)$
Sex	SEX	EMPL		
Rate	RATE	EMPL	$\forall e : empls \bullet e.Rate \geq 4.5$	
HiredBy	DNUM	EMPL		$\forall e : empls \bullet (\exists d : depts \bullet e.HiredBy = d.DNum)$
Dnum	DNUM	DEPT	$\forall d : depts \bullet d.DNum \neq Null \wedge$ $\forall d_1, d_2 : depts \bullet d_1.DNum = d_2.DNum \Leftrightarrow d_1 = d_2$	$\forall e : empls \bullet (\exists d : depts \bullet e.HiredBy = d.DNum) \wedge$ $\forall p : projs \bullet (\exists d : depts \bullet p.CtrlBy = d.DNum)$
ManENum	ENUM	DEPT		$\forall d : depts \bullet (\exists e : empls \bullet e.ENum = d.ManENum)$
DCity	CITY	DEPT		
PNum	PNUM	PROJ	$\forall p : projs \bullet p.PNum \neq Null \wedge$ $\forall p_1, p_2 : projs \bullet p_1.PNum = p_2.PNum \Leftrightarrow p_1 = p_2$	$\forall w : works \bullet (\exists p : projs \bullet p.PNum = w.PNum)$
CtrlBy	DNUM	PROJ		
StartDate	DATE	PROJ	$\forall p : projs \bullet p.EndDate \geq p.StartDate$	
EndDate	DATE	PROJ	$\forall p : projs \bullet p.EndDate \geq p.StartDate$	
PCity	CITY	PROJ		
ENum	ENUM	WORK	$\forall w : works \bullet w.ENum \neq Null \wedge$ $\forall w_1, w_2 : works \bullet (w_1.PNum = w_2.PNum$ $\wedge w_1.ENum = w_2.ENum) \Leftrightarrow w_1 = w_2$	$\forall w : works \bullet (\exists e : empls \bullet e.ENum = w.ENum)$
PNum	PNUM	WORK	$\forall w : works \bullet w.PNum \neq Null \wedge$ $\forall w_1, w_2 : works \bullet (w_1.PNum = w_2.PNum$ $\wedge w_1.ENum = w_2.ENum) \Leftrightarrow w_1 = w_2$	$\forall w : works \bullet (\exists p : projs \bullet p.PNum = w.PNum)$
Hours	HOURS	WORK		

5.2 Transactions specified in Z

A relational database system is a system in which the data is perceived by its user as relations. Data manipulation allows users to communicate with information stored in the system. The user of the system is given facilities to perform a variety of operations on relations including retrieving data from existing relations and updating the contents of the system. In this section, a list of twenty-five transactions that manipulate data in the system are first informally described and then formally specified using Z.

(1) This transaction retrieves full details of all projects that have finished. A project is finished if its *EndDate* attribute contains a date that is earlier that current date.

Proj_End

Ξ DB

lproj! : P PROJ

lproj! = {pj : projs | EndDate < Date()}

(2) This transaction retrieves full details of all employees who earn minimum wage, assuming the minimum wage is $4.25 per hour.

Emp_Min

Ξ DB

lemp! : P EMPL

lemp! = {em : empls | rate = 4.25}

(3) This transaction requests full details of all project assignments of male employees.

All_Work

Ξ DB

lwk! : P WORK

lwk! = {wk : works, em : empls | wk.ENum = em.ENum $\wedge$ em.Sex = Male
• (wk.ENum, wk.PNum, wk.Hours }

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

(4) For a given project p1, this transaction determines the sexes of the employees who work on the projects.

Sex_p

Ξ DB
lsex! : P SEX
lsex! = {em : empls, wk : works | wk.ENum = em.ENum ∧ wk.PNum = p1 • (em.Sex)}

(5) This transaction searches for male employees and display all attributes for each employee found.

Emp_Male

Ξ DB
lemp! : P EMPL
lemp! = {em : empls | Sex = "Male"}

(6) This transaction requests the IDs of all female managers of the departments that are based in New York.

Fem_Mng

Ξ DB
lemp! : P ENUM
lemp! = {em : empls, dpt : depts | dpt.ManENum = em.ENum ∧ em.Sex = "Female"
∧ dpt.DCity = "New York" • (em.ENum)}

(7) This transaction lists the IDs and the corresponding RATEs of the employees who work more than 100 hours on project p2.

Emp_p

Ξ DB
lemp! : P ENUM X RATE
lemp! = {em : empls, wk : works | wk.ENum = em.ENum
∧ wk.PNum = p2 ∧ wk.Hour > 100 • (em.ENum, em.Rate)}

(8) This transaction determines the IDs of the employees who work on the project controlled by the same department that hires them.

Same_Dept

Ξ DB
lemp! : P ENUM
lemp! = {em : empls, pj : projs, wk :works | em.ENum = wk.ENum ∧ wk.PNum = pj.PNum
∧ pj.CtrlBy = em.HiredBy • (em.ENum)}

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

(9) This transaction retrieves the IDs of all departments in New York whose managers' basic rates are higher than 50.

High_Mng

Ξ DB

ldno! : P DNUM

ldno! = {em : empls, dpt : depts | dpt.ManENum = em.ENum ∧ em.Rate > 50
∧ dpt.DCity = "New York" • (dpt.DNum)}

(10) This transaction searches for all departments whose managers are male.

Dept_M

Ξ DB

ldno! : P DEPT

ldno! = {em : empls, dpt : depts | dpt.ManENum = em.ENum ∧ em.Sex = Male
• (dpt.DNum, dpt.ManENum, dpt.DCity)}

(11) This transaction gets the IDs of all projects in New York that are controlled by departments whose managers are female.

Same_Dept

Ξ DB

lpno! : P PNUM

lpno! = {em : empls, pj : projs, dpt : depts | em.ENum = dpt.ManENum
∧ dpt.DNum = pj.CtrlBy ∧ pj.City = "New York" ∧ em.Sex = Female • (pj.PNum)}

(12) This transaction lists the IDs and SEX of the employees whose basic rates are under 10.

Emp_Sex

Ξ DB

lemp! : P ENUM X SEX

lemp! = {em : empls | Rate < 10 • (ENum, Sex)}

(13) This transaction requests an input value which indicates the ID of an employee. The transaction lists all project numbers and the corresponding hours the employee has worked on.

Proj_e

Ξ DB

e? : ENUM

lpj! : P PNUM X HOUR

lpj! = {wk : works | wk.ENum = e? • (PNum, Hour)}

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

(14) This transaction lists the IDs of the departments that are based in a city.

Dept_c

Ξ DB
c? : CITY
ldno! : P DNUM

ldno! = {dpt : depts | dpt.City = c? • (DNum)}

(15) This transaction searches for the employees who are hired after a given project has started. The ID of the project is entered by the user. In addition, the employees retrieved are assigned to work on the input project. The transaction reveals two attributes of the employees: ID and hourly Rate.

Emp_h

Ξ DB
p? : PNUM
lemp! : P ENUM X RATE

lemp! = {em : empls, wk : works, pj : projs | pj.PNum = p? ∧ wk.ENum = em.ENum
∧ wk.PNum = pj.PNum ∧ pj.StartDate < em.HiredDate • (ENum, Rate)}

(16) This transaction retrieves all details of the projects that are in the same city as the departments that control them.

Proj_city

Ξ DB
lpj! : P PROJ

lpj! = {pj : projs, dpt : depts | pj.CtrlBy = dpt.DNum ∧ pj.City = dpt.City
• (PNum, StartDate, EndDate, City, CtrlBy)}

(17) This transaction requests an input value from the user. The input value indicates the ID of the department of interest. The transaction then searches for the projects that are controlled by the input department before its new manager was hired.

P_Ctrl_d

Ξ DB
d? : DNUM
lpj! : P PROJ

lpj! = {em : empls, dpt : depts, pj : projs | pj.CtrlBy = d? ∧ dpt.ManENum = em.ENum
∧ dpt.DNum = pj.CtrlBy ∧ pj.EndDate < em.HiredDate
• (PNum, City, CtrlBy, StartDate,EndDate)}

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

(18) This transaction lists full details of the departments in New York.

Dept_NY

Ξ DB

lproj! : P PROJ

lproj! = {pj : projs | City = "New York" }

(19) This transaction is looking for full details of the projects that are controlled by a given department. The department ID is determined based on an input value.

Proj_d

Ξ DB

d? : DNUM

lpj! : P PROJ

lpj! = {pj : projs | CtrlBy = d?}

(20) This transaction searches for the employees who work for the projects in New York. Only the IDs and the hourly rate of the employees will be displayed.

E_P_NY

Ξ DB

lemp! : P ENUM X RATE

lemp! = {pj : projs, em : empl, wk : works | pj.PNum = wk.PNum $\wedge$ pj.City = "New York" $\wedge$ em.ENum = wk.ENum $\wedge$ em.Sex = "Female" • (Enum, Rate)}

(21) This transaction allows user to enter two dates and searches for the employees who are hired between the two dates.

Emp_Date

Ξ DB

d1?, d2? : DATE

lemp! : P EMPL

lemp! = {em : empls | HiredDate $\geq$ d1? $\wedge$ HiredDate $\leq$d2?}

(22) This transaction searches for the employees who work less than 10 hours on a given project. The user provides the ID for the project of interest. Only the IDs of the employees retrieved will be displayed.

Emp_Date

Ξ DB

p? : PNUM

lem! : P ENUM

lem! = {wk : works | PNum = p? • (wk.ENum)}

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

(23) For the projects that are controlled by departments in New York, this transaction lists the ID for the departments and the hourly rate of the managers of the departments.

```
┌──── Dept_P_NY ─────────────────────────────────────────
│ Ξ DB
│ lemp! : P DNUM X RATE
├────────────────────────────────────────────────────────
│ lemp! = {pj : projs, em : empl, dpt : works | pj.CtrlBy = dpt.DNum ∧ pj.City = "New York"
│               ∧ em.ENum = dpt.ManENum • (dpt.DNum, em.Rate)}
└────────────────────────────────────────────────────────
```

(24) This transaction retrieves the IDs and hours of the projects that are worked by female employees.

```
┌──── Proj_female ───────────────────────────────────────
│ Ξ DB
│ lemp! : P PNUM X HOURS
├────────────────────────────────────────────────────────
│ lemp! = {em : empl, wk : works | pj.ENum = em.ENum ∧ em.Sex = "Female"
│                                         • (wk.PNum, wk.Hour)}
└────────────────────────────────────────────────────────
```

(25) This transaction searches for all male employees in the database. Only the IDs and the associated hourly rates for the employees will be returned.

```
┌──── Rate_Male ──────────────────────────
│ Ξ DB
│ lemp! : P ENUM X RATE
├─────────────────────────────────────────
│ lemp! = {em : empls | em.Sex = "Male" • (em.ENum, em.Rate)}
└─────────────────────────────────────────
```

5.3 Reusability Analysis Phase

After the transactions are specified using Z, the resulting specification is treated as the input to RRM. The first phase of RRM utilizes the clustering analysis method to identify groups of transactions according to their semantics functionality. There are three main steps in the reusability analysis phase: extracting descriptor sets for transactions, calculating similarity indices among transactions according to the

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

descriptor sets, and identifying clustering structure. Sections 5.3.1, 5.3.2 and 5.3.3 present the result of the case study for each of these steps respectively.

5.3.1 Descriptor sets

The descriptor set of any given transaction consists of a number of pairs of <descriptor, value>. The descriptor reflects one type of basic operation requested by the transaction. The corresponding value indicates the level of effort involved in the operation. Since the specification of the transactions follow the predicate patterns defined in Section 3.1, the descriptor set of a given transaction is extracted by parsing the corresponding Z schema. Section 3.2.1 used several examples to illustrate the process of extracting a descriptor set from a Z schema. Here, the resulting descriptor sets of the transactions specified in section 5.2 are listed in Table 5.2.

5.3.2 Similarity indices

The similarity function was defined in Section 3.3.1 as the ratio of the cardinality of the intersection over the cardinality of the union of the descriptor sets that correspond to two transactions. Definitions of the intersection and union of two given descriptor sets were given in Sections 3.2.2 and 3.2.3, respectively. In Section 3.2.4, an algorithm was defined to compute the cardinality of a given descriptor set. The algorithm is applied to the pair-wise intersections and unions of the descriptor sets in Table 5.3. Table 5.4 lists the cardinalities of the pair wise intersections among the descriptor sets extracted in Section 5.2, while the cardinalities of the pair wise unions among the descriptor sets extracted in Section 5.2 are listed in Table 5.5. The resulting similarity indices among the transactions are presented in Table 5.6.

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

Table 5.3 Descriptor sets for transactions in section 5.2

Transaction	Descriptor
1	<output, old><select, 1>
2	<output, old><select, 1>
3	<output, old><select, 1><join, 2>
4	<output, new><select, 1><join, 2>
5	<output, old><select, 1>
6	<output, new><select, 2><join, 2>
7	<output, new><select, 1><join, 2>
8	<output, new><join, 3>
9	<output, new><select, 2><join, 2>
10	<output, old><select, 1><join, 2>
11	<output, new><select, 2><join, 3>
12	<output, new><select, 1>
13	<input, value><output, new><select, 1>
14	<input, value><output, new><select, 1>
15	<input, value><output, new><select, 1><join, 3>
16	<output, old><join, 2>
17	<input, value><output, old><select, 1><join, 3>
18	<output, old><select, 1>
19	<input, value><output, old><select, 1>
20	<output, new><select, 2><join, 3>
21	<input, value><output,old><select, 1>
22	<input, value><output, new><select, 1>
23	<output, new><select, 1><join, 3>
24	<output, new><select, 1><join, 2>
25	<output, new><select, 1>

5.3.3 Cluster structure

The clustering analysis method described in Section 3.3 is applied to the similarity matrix in Table 5.5 by starting with clusters of singleton transactions and repeatedly searching and merging the two most similar clusters into one. The process continues until the similarity chosen falls under 0.5 implying that the percentage of common functionality between transactions in any two clusters falls below 50 percent. The history of merging is presented in Figure 5.1.

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

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

Table 5.4 Cardinalities of pair-wise intersections

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	4	4	4	3	4	3	3	0	3	4	3	3	3	3	3	1	4	4	4	3	4	3	3	3	3
2	4	4	4	3	4	3	3	0	3	4	3	3	3	3	3	1	4	4	4	3	4	3	3	3	3
3	4	4	12	11	4	11	11	8	11	12	11	3	3	3	11	9	12	4	4	11	4	3	11	11	3
4	3	3	11	12	3	12	12	9	12	11	12	4	4	4	12	8	11	3	3	12	3	4	12	12	4
5	4	4	4	3	4	3	3	0	3	4	3	3	3	3	3	1	4	4	4	3	4	3	3	3	3
6	3	3	11	12	3	15	12	9	15	11	15	4	4	4	12	8	11	3	3	15	3	4	12	12	4
7	3	3	11	12	3	12	12	9	12	11	12	4	4	4	12	8	11	3	3	12	3	4	12	12	4
8	0	0	8	9	0	9	9	13	9	8	13	1	1	1	13	8	12	0	0	13	0	1	13	9	1
9	3	3	11	12	3	15	12	9	15	11	15	4	4	4	12	8	11	3	3	15	3	4	12	12	4
10	4	4	12	11	4	11	11	8	11	12	11	3	3	3	11	9	12	4	4	11	4	3	11	11	3
11	3	3	11	12	3	15	12	13	15	11	19	4	4	4	16	8	15	3	3	19	3	4	16	12	4
12	3	3	3	4	3	4	4	1	4	3	4	4	4	4	4	10	3	3	3	4	3	4	4	4	4
13	3	3	3	4	3	4	4	1	4	3	4	4	6	6	6	0	5	3	5	4	5	6	4	4	4
14	3	3	3	4	3	4	4	1	4	3	4	4	6	6	6	0	5	3	5	4	5	6	4	4	4
15	3	3	11	12	3	12	12	13	12	11	16	4	6	6	18	8	17	3	5	16	5	6	16	12	4
16	1	1	9	8	1	8	8	8	8	9	8	0	0	0	8	9	9	1	1	8	1	0	8	8	0
17	4	4	12	11	4	11	11	12	11	12	15	3	5	5	17	9	18	4	6	15	6	5	15	11	3
18	4	4	4	3	4	3	3	0	3	4	3	3	3	3	3	1	4	4	4	3	4	3	3	3	3
19	4	4	4	3	4	3	3	0	3	4	3	3	5	5	5	1	6	4	6	3	6	5	3	3	3
20	3	3	11	12	3	15	12	13	15	11	19	4	4	4	16	8	15	3	3	19	3	4	16	12	4
21	4	4	4	3	4	3	3	0	3	4	3	3	5	5	5	1	6	4	6	3	6	5	3	3	3
22	3	3	3	4	3	4	4	1	4	3	4	4	6	6	6	0	5	3	5	4	5	6	4	4	4
23	3	3	11	12	3	12	12	13	12	11	16	4	4	4	16	8	15	3	3	16	3	4	16	12	4
24	3	3	11	12	3	12	12	9	12	11	12	4	4	4	12	8	11	3	3	12	3	4	12	12	4
25	3	3	3	4	3	4	4	1	4	3	4	4	4	4	4	0	3	3	3	4	3	4	4	4	4

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

Table 5.5 Cardinalities of pair-wise unions

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	4	4	12	13	4	16	13	17	16	12	20	5	7	7	19	12	18	4	6	20	6	7	17	13	5
2	4	4	12	13	4	16	13	17	16	12	20	5	7	7	19	12	18	4	6	20	6	7	17	13	5
3	12	12	12	13	12	16	13	17	16	12	20	13	15	15	19	12	18	12	14	20	14	15	17	13	13
4	13	13	13	12	13	15	12	16	15	13	19	12	14	14	18	13	19	13	15	19	15	14	16	12	12
5	4	4	12	13	4	16	13	17	16	12	20	5	7	7	19	12	18	4	6	20	6	7	17	13	5
6	16	16	16	15	16	15	15	19	15	16	19	15	17	17	21	16	22	16	18	19	18	17	19	15	15
7	13	13	13	12	13	15	12	16	15	13	19	12	14	14	18	13	19	13	15	19	15	14	16	12	12
8	17	17	17	16	17	19	16	13	19	17	19	16	18	18	18	14	19	17	19	19	19	18	16	16	16
9	16	16	16	15	16	15	15	19	15	16	19	15	17	17	21	16	22	16	18	19	18	17	19	15	15
10	12	12	12	13	12	16	13	17	16	12	20	13	15	15	19	12	18	12	14	20	14	15	17	13	13
11	20	20	20	19	20	19	19	19	19	20	19	19	21	21	21	20	22	20	22	19	22	21	19	19	19
12	5	5	13	12	5	15	12	16	15	13	19	4	6	6	18	13	19	5	7	19	7	6	16	12	4
13	7	7	15	14	7	17	14	18	17	15	21	6	6	6	18	15	19	7	7	21	7	6	18	14	6
14	7	7	15	14	7	17	14	18	17	15	21	6	6	6	18	15	19	7	7	21	7	6	18	14	6
15	19	19	19	18	19	21	18	18	21	19	21	18	18	18	18	19	19	19	19	21	19	18	18	18	18
16	12	12	12	13	12	16	13	14	16	12	20	13	15	15	19	9	18	12	14	20	14	15	17	13	13
17	18	18	18	19	18	22	19	19	22	18	22	19	19	19	19	18	18	18	18	22	18	19	19	19	19
18	4	4	12	13	4	16	13	17	16	12	20	5	7	7	19	12	18	4	6	20	6	7	17	13	5
19	6	6	14	15	6	18	15	19	18	14	22	7	7	7	19	14	18	6	6	22	6	7	19	15	7
20	20	20	20	19	20	19	19	19	19	20	19	19	21	21	21	20	22	20	22	19	22	21	19	19	19
21	6	6	14	15	6	18	15	19	18	14	22	7	7	7	19	14	18	6	6	22	6	7	19	15	7
22	7	7	15	14	7	17	14	18	17	15	21	6	6	6	18	15	19	7	7	21	7	6	18	14	6
23	17	17	17	16	17	19	16	16	19	17	19	16	18	18	18	17	19	17	19	19	19	18	16	16	16
24	13	13	13	12	13	15	12	16	15	13	19	12	14	14	18	13	19	13	15	19	15	14	16	12	12
25	5	5	13	12	5	15	12	16	15	13	19	4	6	6	18	13	19	5	7	19	7	6	16	12	4

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

Table 5.6 Similarity matrix

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
1	1.00	1.00	0.33	0.23	1.00	0.19	0.23	0.00	0.19	0.33	0.15	0.60	0.43	0.43	0.16	0.08	0.22	1.00	0.67	0.15	0.67	0.43	0.18	0.23	0.60
2	1.00	1.00	0.33	0.23	1.00	0.19	0.23	0.00	0.19	0.33	0.15	0.60	0.43	0.43	0.16	0.08	0.22	1.00	0.67	0.15	0.67	0.43	0.18	0.23	0.60
3	0.33	0.33	1.00	0.85	0.33	0.69	0.85	0.47	0.69	1.00	0.55	0.23	0.20	0.20	0.58	0.75	0.67	0.33	0.29	0.55	0.29	0.20	0.65	0.85	0.23
4	0.23	0.23	0.85	1.00	0.23	0.80	1.00	0.56	0.80	0.85	0.63	0.33	0.29	0.29	0.67	0.62	0.58	0.23	0.20	0.63	0.20	0.29	0.75	1.00	0.33
5	1.00	1.00	0.33	0.23	1.00	0.19	0.23	0.00	0.19	0.33	0.15	0.60	0.43	0.43	0.16	0.08	0.22	1.00	0.67	0.15	0.67	0.43	0.18	0.23	0.60
6	0.19	0.19	0.69	0.80	0.19	1.00	0.80	0.47	1.00	0.69	0.79	0.27	0.24	0.24	0.57	0.50	0.50	0.19	0.17	0.79	0.17	0.24	0.63	0.80	0.27
7	0.23	0.23	0.85	1.00	0.23	0.80	1.00	0.56	0.80	0.85	0.63	0.33	0.29	0.29	0.67	0.62	0.58	0.23	0.20	0.63	0.20	0.29	0.75	1.00	0.33
8	0.00	0.00	0.47	0.56	0.00	0.47	0.56	1.00	0.47	0.47	0.68	0.06	0.06	0.06	0.72	0.57	0.63	0.00	0.00	0.68	0.00	0.06	0.81	0.56	0.06
9	0.19	0.19	0.69	0.80	0.19	1.00	0.80	0.47	1.00	0.69	0.79	0.27	0.24	0.24	0.57	0.50	0.50	0.19	0.17	0.79	0.17	0.24	0.63	0.80	0.27
10	0.33	0.33	1.00	0.85	0.33	0.69	0.85	0.47	0.69	1.00	0.55	0.23	0.20	0.20	0.58	0.75	0.67	0.33	0.29	0.55	0.29	0.20	0.65	0.85	0.23
11	0.15	0.15	0.55	0.63	0.15	0.79	0.63	0.68	0.79	0.55	1.00	0.21	0.19	0.19	0.76	0.40	0.68	0.15	0.14	1.00	0.14	0.19	0.84	0.63	0.21
12	0.60	0.60	0.23	0.33	0.60	0.27	0.33	0.06	0.27	0.23	0.21	1.00	0.67	0.67	0.22	0.00	0.16	0.60	0.43	0.21	0.43	0.67	0.25	0.33	1.00
13	0.43	0.43	0.20	0.29	0.43	0.24	0.29	0.06	0.24	0.20	0.19	0.67	1.00	1.00	0.33	0.00	0.26	0.43	0.71	0.19	0.71	1.00	0.22	0.29	0.67
14	0.43	0.43	0.20	0.29	0.43	0.24	0.29	0.06	0.24	0.20	0.19	0.67	1.00	1.00	0.33	0.00	0.26	0.43	0.71	0.19	0.71	1.00	0.22	0.29	0.67
15	0.16	0.16	0.58	0.67	0.16	0.57	0.67	0.72	0.57	0.58	0.76	0.22	0.33	0.33	1.00	0.42	0.89	0.16	0.26	0.76	0.26	0.33	0.89	0.67	0.22
16	0.08	0.08	0.75	0.62	0.08	0.50	0.62	0.57	0.50	0.75	0.40	0.00	0.00	0.00	0.42	1.00	0.50	0.08	0.07	0.40	0.07	0.00	0.47	0.62	0.00
17	0.22	0.22	0.67	0.58	0.22	0.50	0.58	0.63	0.50	0.67	0.68	0.16	0.26	0.26	0.89	0.50	1.00	0.22	0.33	0.68	0.33	0.26	0.79	0.58	0.16
18	1.00	1.00	0.33	0.23	1.00	0.19	0.23	0.00	0.19	0.33	0.15	0.60	0.43	0.43	0.16	0.08	0.22	1.00	0.67	0.15	0.67	0.43	0.18	0.23	0.60
19	0.67	0.67	0.29	0.20	0.67	0.17	0.20	0.00	0.17	0.29	0.14	0.43	0.71	0.71	0.26	0.07	0.33	0.67	1.00	0.14	1.00	0.71	0.16	0.20	0.43
20	0.15	0.15	0.55	0.63	0.15	0.79	0.63	0.68	0.79	0.55	1.00	0.21	0.19	0.19	0.76	0.40	0.68	0.15	0.14	1.00	0.14	0.19	0.84	0.63	0.21
21	0.67	0.67	0.29	0.20	0.67	0.17	0.20	0.00	0.17	0.29	0.14	0.43	0.71	0.71	0.26	0.07	0.33	0.67	1.00	0.14	1.00	0.71	0.16	0.20	0.43
22	0.43	0.43	0.20	0.29	0.43	0.24	0.29	0.06	0.24	0.20	0.19	0.67	1.00	1.00	0.33	0.00	0.26	0.43	0.71	0.19	0.71	1.00	0.22	0.29	0.67
23	0.18	0.18	0.65	0.75	0.18	0.63	0.75	0.81	0.63	0.65	0.84	0.25	0.22	0.22	0.89	0.47	0.79	0.18	0.16	0.84	0.16	0.22	1.00	0.75	0.25
24	0.23	0.23	0.85	1.00	0.23	0.80	1.00	0.56	0.80	0.85	0.63	0.33	0.29	0.29	0.67	0.62	0.58	0.23	0.20	0.63	0.20	0.29	0.75	1.00	0.33
25	0.60	0.60	0.23	0.33	0.60	0.27	0.33	0.06	0.27	0.23	0.21	1.00	0.67	0.67	0.22	0.00	0.16	0.60	0.43	0.21	0.43	0.67	0.25	0.33	1.00

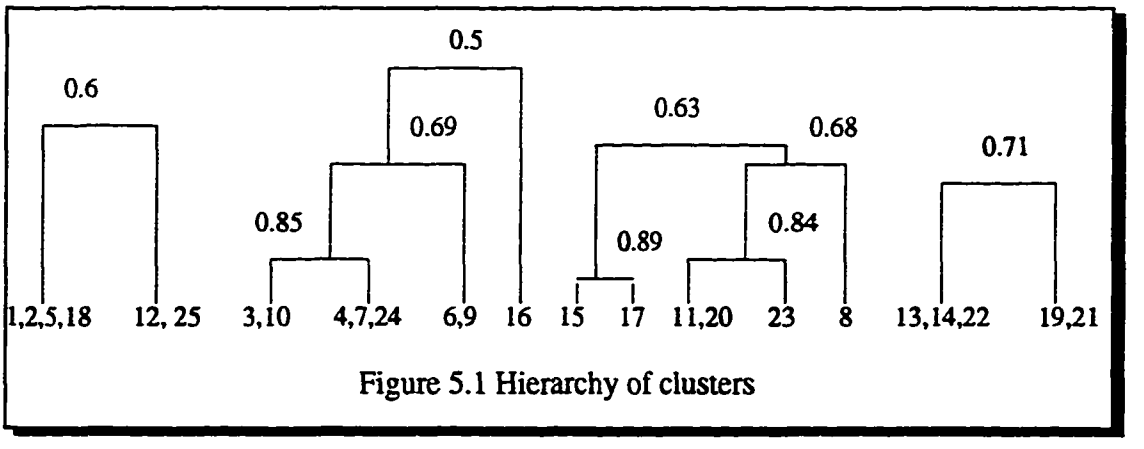

Figure 5.1 Hierarchy of clusters

As a result, there are four final clusters of transactions:

Cluster1: contains transactions 1, 2, 5, 12, 18, and 25 with 0.60 as the minimum similarity preserved within the cluster;

Cluster2: contains transactions 3, 4, 6, 7, 9, 10, 16, and 24 with 0.50 as the minimum similarity preserved within the cluster;

Cluster3: contains transactions 8, 11, 15, 17, 20, and 23 with 0.63 as the minimum similarity preserved within the cluster;

Cluster4: contains transactions 13, 14, 19, 21, and 22 with 0.71 as the minimum similarity preserved within the cluster.

Table 5.1 is extended by an additional column recording the cluster number to which each transaction belongs. The extended table is shown in Table 5.7

5.4 Reusability Realization Phase

Section 5.3 described the analysis of the specifications of the transactions that were presented in Section 5.2. A clustering structure was derived where transactions that are similar in function were grouped together into the same cluster. This section

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

Table 5.7 Transaction dictionary (1)

Transaction	Descriptor	Cluster
1	<output, old><select, 1>	1
2	<output, old><select, 1>	1
3	<output, old><select, 1><join, 2>	2
4	<output, new><select, 1><join, 2>	2
5	<output, old><select, 1>	1
6	<output, new><select, 2><join, 2>	2
7	<output, new><select, 1><join, 2>	2
8	<output, new><join, 3>	3
9	<output, new><select, 2><join, 2>	2
10	<output, old><select, 1><join, 2>	2
11	<output, new><select, 2><join, 3>	3
12	<output, new><select, 1>	1
13	<input, value><output, new><select, 1>	4
14	<input, value><output, new><select, 1>	4
15	<input, value><output, new><select, 1><join, 3>	3
16	<output, old><join, 2>	2
17	<input, value><output, old><select, 1><join, 3>	3
18	<output, old><select, 1>	1
19	<input, value><output, old><select, 1>	4
20	<output, new><select, 2><join, 3>	3
21	<input, value><output,old><select, 1>	4
22	<input, value><output, new><select, 1>	4
23	<output, new><select, 1><join, 3>	3
24	<output, new><select, 1><join, 2>	2
25	<output, new><select, 1>	1

illustrates the reusability realization phase of RRM. The first step of this phase is to extract generic requirements for each cluster(Section 5.4.1). Section 5.4.2 illustrates the step of generating reusable frameworks for clusters to accomplish the requirements. In addition, the transactions are implemented in C in Section 5.4.3.

5.4.1 Generic requirements

Based on the descriptor sets of the transactions that are grouped into the same cluster, generic requirements are extracted according to the algorithms given in Section

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

4.1. Since the sizes of the clusters are not large, one framework for each cluster is sufficient for reuse for the implementation of individual transactions. However, we still extract the minimum and maximum requirements for the purpose of illustration. The generic requirements are automatically extracted by REST and represented under the same measurement space as that for the transactions, i.e. descriptor sets. The results are listed in Table 5.7.

Table 5.8 Generic requirements for clusters

Cluster	Minimum Requirements	Maximum Requirements
1	<output old><select 1>	<output new><select 1>
2	<output old><join 2>	<output new><select 2><join 2>
3	<output old><join 3>	<input value><output new><select 2><join 3>
4	<input value><output old><select 1>	<input value><output new><select 1>

5.4.2 Reusable frameworks

For each cluster, the minimum and maximum generic requirements were extracted in Section 5.4.1. In this section, reusable templates are generated to accomplish these requirements. For each template, there are three parts: interface, local variables and body. The interface of a template indicates the name of the template, which is a combination of the cluster to which the template belongs and the type of generic requirements it accomplishes, input parameters, and output parameters. The local variables correspond to the intermediate output relation and input information. The body of the template shows an implementation of the requirements.

From Table 5.8, the descriptor set of the minimum requirement for Cluster 1 includes two <descriptor, value> pairs. Figures 5.2 presents the framework for the

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

minimum requirement of Cluster 1. The figure is followed by a discussion of how the framework is generated.

```
Interface:
        Name:           Cluster1_Min
        Input parameters:
                select: relation_1, size_1
        Output parameters:    None
Local variables:
        Index variable:       int i;
        Resulting relation:   struct existing_relation result[];
                              int size_result;
Body:
        size_result = 0;        /* initialization */
        for (i = 0; i < size_1; i++)            /* search relations_1 */
            if (<select_condition> on relation_1)
            {   assigning tuple to result;
                size_result++;      }
        print_relation(result, size_result);
```

Figure 5.2 Framework for the minimum requirements of cluster 1

The first <descriptor, value> pair is <output, old>, which indicates that there is an intermediate output relation requested that takes the structure of an existing relation. Therefore, in the local variable section of the framework, an array of struct needs to be declared where the type of the struct is an existing relation. In addition, an integer variable needs to be declared to store the size of the intermediate relation at the end of the transaction. The second <descriptor, value> pair is <select, 1> which indicates that a filter needs to be applied to one relation. Thus, when designing the interface of the framework, two input parameters need to be listed: a relation and its corresponding size. The body of the framework presents the steps required to accomplish the retrieval. Since the only operation that the framework is dealing with is *select*, a FOR loop is set

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

up to allow all records in the input relation to become candidate records for the new relation. The <S_condition> on the input relation serves as a filter to select records that satisfy certain criteria. A built-in function *print_relation* is called to print the result.

The maximum and minimum requirements for Cluster 1 are very similar. They both include <select, 1> as a member. Therefore, the input parameters contain both the relation to be searched and the size of the relation. In addition, the body of the framework needs to have a FOR loop which applies a filter to the input relation. The only difference is that the intermediate relation now takes a structure that is not the same as any existing relation. Therefore, a template needs to be set up in the local variable section of the framework to allow reusers of this framework to fill in detail information about the fields (attributes) and the domains from which the fields draw values. Figures 5.3 presents the framework for the maximum requirements of Cluster 1.

```
Interface:
        Name:           Cluster1_Max
        Input parameters:
                select: relation_1, size_1
        Ouput parameters:       None
Local variables:
        Index variable:         int i;
        struct definition:struct relation_new {
                                        [domain  field;]+
                                };
        Resulting relation:     struct relation_new result[];
                                int size_result;
Body:
        size_result = 0; /* initialization */
        for (i = 0; i < size_1; i++)                /* search relations_1 */
            if (<select_condition> on relation_1)
            {   assigning tuple to result
                size_result++;      }
        print_relation(result, size_result);
```

Figure 5.3 Framework for the maximum requirements of cluster 1

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

The minimum requirement for Cluster 2 includes <output, old> as a member of its descriptor set, as did Cluster 1. Thus the resulting relation section of the local varible part in the framework is the same as that shown in Figure 5.2. In addition, <join, 2> is the other member of the descriptor set which indicates that two relations and their corresponding size variables need to be included as input parameters for the framework. The join operation requires a coordination between the two input relations. A nested FOR loop is necessary in the body of the framework to allow searching the two input relations simultaneously. Two records, one from each input relation, are compared based on the Join_condition. If a match is found, a new tuple for the resulting relation takes the appropriate value from the two records and is added to the resulting relation.

```
Interface:
        Name:           Cluster2_Min
        Input parameters:
                join:   relation_1, size_1, relation_2, size_2
        Ouput parameters:   None
Local variables:
        Index variable:         int i, j;
        Resulting relation:     struct existing_relation result[];
                                int size_result;
Body:
        size_result = 0;        /* initialization */
        for (i = 0; i < size_1; i++)            /* search relation_1 */
            for (j = 0; j < size_2; i++)        /* search relation_2 */
                if (<J_cond.> match between relation_1 and relation_2)
                {
                        assigning tuple to result;
                        size_result++;
                }
        print_relation(result, size_result);
```

Figure 5.4 Framework for the minimum requirements of cluster 2

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

The framework for the maximum requirement of Cluster 2 is shown in Figure 5.5. The <output, new> pair in the descriptor set of the maximum requirement for Cluster 2 is treated the same as that in the descriptor set of the maximum requirement for Cluster 1. A template is included in the local variable section of the framework to allow the potential reuser of the framework to fill in detail information about what

```
Interface:
        Name:           Cluster2_Max
        Input parameters:
                select: relation_1, size_1, relation_2, size_2
                join:   relation_1, size_1, relation_2, size_2
        Ouput parameters:    None
Local variables:
        Index variable:      int i;
        struct definition:   struct relation_new {
                                    [domain  field;]+
                             };
        Resulting relation:  struct relation_new result[];
                             int size_result;
Body:
        size_result = 0;          /* initialization */
        for (i = 0; i < size_1; i++)          /* search relations_1 */
            if (<S_cond_1> on relation_1)
                for (j = 0; j < size_2; j++)
                    if (<S_cond_2> on relation_2)
                        and (<J_cond.> matches relation_1 and relation_2)
                        {    assigning tuple to result
                             size_result++;
                        }
        print_relation(result, size_result);
```

Figure 5.5 Framework for the maximum requirements of cluster 2

fields are included in the resulting relation and what domains provide values for the fields. The <join, 2> pair in the descriptor set of the maximum requirement for Cluster 2 is treated the same way as that in the descriptor set of the minimum requirement for

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

Cluster 2. A nested FOR loop is set up in the body of the framework to join the records in the two input relations together. In addition, the descriptor set of the maximum requirement for Cluster 2 contains <select, 2> as one of its members. The pair indicates that two filters need to be applied to the input relations, one filter for each relation. Therefore, before two records, each from one input relation, are compared against each other, IF statements need to be used to see if the two records satisfy certain criteria.

The frameworks for the minimum and maximum requirements for Clusters 3 and 4 are listed in Figures 5.6 to 5.9 respectively. The generation of these frameworks follows a similar process as discussed for the previous frameworks.

```
Interface:
        Name:           Cluster3_Min
        Input parameters:
                join:   relation_1, size_1, relation_2, size_2, relation_3, size_3
        Ouput parameters:       None

Local variables:
        Index variable:         int i, j, k;
        Resulting relation:     struct existing_relation result[];
                                int size_result;

Body:
        size_result = 0;        /* initialization */
        for (i = 0; i < size_1; i++)            /* search relation_1 */
            for (j = 0; j < size_2; i++)        /* search relation_2 */
                if (<join_condition> match between relation_1 and relation_2)
                    for (k = 0; k < size_3; k++)        /* search relation_3 */
                        if (<J_con_2> match between rel_3 to rel_(1 and/or 2))
                            {
                                assigning tuple to result;
                                size_result++;
                            }
        print_relation(result, size_result);
```

Figure 5.6 Framework for the minimum requirements of cluster 3

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

Interface:

Name: Cluster3_Max

Input parameters:

select: relation_1, size_1, relation_2, size_2

join: relation_1, size_1, relation_2, size_2, relation_3, size_3

Ouput parameters: None

Local variables:

Index variable: int i, j, k;

Input media: datatype value;

struct definition: struct relation_new
{
[domain field;]+
};

Resulting relation: struct relation_new result[];
int size_result;

Body:

```
Get_value(&value);
size_result = 0;      /* initialization */
for (i = 0; i < size_1; i++)                    /* search relation_1 */
  if (<S_cond> on relation_1)
    for (j = 0; j < size_2; i++)                /* search relation_2 */
      if (<J_cond> between relation_1 and relation_2)
                  and (<S_cond> on relation_2)
          for (k = 0; k < size_3; k++)  /* search relation_3 */
            if (<J_cond> between relation_3
                  and relation_(1 and/or 2))
            {
                  assigning tuple to result;
                  size_result++;
            }
print_relation(result, size_result);
```

Figure 5.7 Framework for the maximum requirements of cluster 3

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

```
Interface:
        Name:           Cluster4_Min
        Input parameters:
                select: relation_1, size_1
        Ouput parameters:    None
Local variables:
        Index variable:      int i;
        Input media:         datatype value;
        Resulting relation:  struct existing_relation result[];
                             int size_result;
Body:
        Get_value(&value);   /* Get input information */
        size_result = 0;     /* initialization */
        for (i = 0; i < size_1; i++)          /* search relations_1 */
            if (<select_condition> on relation_1)
            {   assigning tuple to result;
                size_result++;      }
        print_relation(result, size_result);
```

Figure 5.8 Framework for the minimum requirements of cluster 4

```
Interface:
        Name:           Cluster4_Max
        Input parameters:
                select: relation_1, size_1
        Ouput parameters:    None
Local variables:
        Index variable:      int i;
        Input media:         datatype value;
        struct definition:   struct relation_new {
                                    [domain  field;]+
                             };
        Resulting relation:  struct relation_new result[];
                             int size_result;
Body:
        Get_value(&value);   /* Get input information */
        size_result = 0;     /* initialization */
        for (i = 0; i < size_1; i++)          /* search relations_1 */
            if (<select_condition> on relation_1)
            {   assigning tuple to result;
                size_result++;      }
        print_relation(result, size_result);
```

Figure 5.9 Framework for the maximum requirements of cluster 4

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

5.4.3 Implementation of transactions

At this stage, a repository of frameworks that are both useful and usable for the implementation of individual transactions has been developed. The transaction dictionary, shown in Table 5.7, reveals the clusters to which the transactions belong. The candidate frameworks are those that are designed for the minimum and maximum requirements of the same cluster. For each transaction, REST computes the effort that is necessary to implement the transaction by modifying each framework of the same cluster. The computation is conducted according to the algorithm in Section 4.3.1. Table 5.7 is extended to Table 5.9 by including the results. The fourth column lists the modification effort from the framework for the minimum requirement. The modification effort from the framework for the maximum requirment is listed in the fifth column. A recommendation for the optimal framework, the framework that requires the smaller estimated modification effort, is provided in the last column. For any given transaction, no retrieval time for the candidate framework is necessary.

Three transactions are chosen to illustrate the adaptation process for cases that require only substitution (transaction 1), both substitution and addition (transaction 3), and both substitution and deletion (transaction 4).

Transaction 1 requests full details of all finished projects. It was specified using Z in section 5.2 and is again listed for convenience to the reader:

Proj_End

Ξ DB

lproj! : $\mathbb{P}$ PROJ

lproj! = {pj : projs | EndDate < Date()}

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

Table 5.9 Transaction dictionary (2)

	Descriptor	Clus.	Dist Min	Dist Max	Choice
1	<output, old><select, 1>	1	0	2	C1_Min
2	<output, old><select, 1>	1	0	2	C1_Min
3	<output, old><select, 1><join, 2>	2	3	5	C2_Min
4	<output, new><select, 1><join, 2>	2	5	3	C2_Max
5	<output, old><select, 1>	1	0	2	C1_Min
6	<output, new><select, 2><join, 2>	2	8	0	C2_Max
7	<output, new><select, 1><join, 2>	2	5	3	C2_Max
8	<output, new><join, 3>	3	2	7	C3_Min
9	<output, new><select, 2><join, 2>	2	8	0	C2_Max
10	<output, old><select, 1><join, 2>	2	3	5	C2_Min
11	<output, new><select, 2><join, 3>	3	8	1	C3_Max
12	<output, new><select, 1>	1	2	0	C1_Max
13	<input, value><output, new><select, 1>	4	2	0	C4_Max
14	<input, value><output, new><select, 1>	4	2	0	C4_Max
15	<input, value><output, new><select, 1><join, 3>	3	6	3	C3_Max
16	<output, old><join, 2>	2	0	8	C2_Min
17	<input, value><output, old><select, 1><join, 3>	3	4	5	C3_Min
18	<output, old><select, 1>	1	0	2	C1_Min
19	<input, value><output, old><select, 1>	4	0	2	C4_Min
20	<output, new><select, 2><join, 3>	3	8	1	C3_Max
21	<input, value><output,old><select, 1>	4	0	2	C4_Min
22	<input, value><output, new><select, 1>	4	2	0	C4_Max
23	<output, new><select, 1><join, 3>	3	5	4	C3_Max
24	<output, new><select, 1><join, 2>	2	5	3	C2_Max
25	<output, new><select, 1>	1	2	0	C1_Max

The descriptor set of the transaction, {<output, old>, <select, 1>}, is the same as that of the minimum requirement for Cluster 1. Therefore, as suggested by Table 5.8, the implementation of the transaction can reuse framework Cluster1_Min. The first step of adaptation is to determine the function prototype. The name of the schema is used as the name of the function. The framework indicates that there is one pair of input parameters: the relation to be searched by the *select* operation and its corresponding

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

size indicator. The Z schema reveals that the name of the relations is *PROJECT*. So, the input parameters are an array of struct *project* and an integer *size_proj*. There is no output parameter for the function, therefore, the return type of the function is *void*. The next step is to determine the local variables necessary. The value corresponds to the descriptor *output* is *old* which indicates that the resulting relation takes the structure of an existing relation. The name of the existing_relation needs to be substituted in the framework. From the schema, we identify that the name of the existing relation is *PROJECT*. The body of the framework contains a FOR loop which searches through the relation and only assigns tuples that satisfy the select condition to the resulting relation. The schema indicates that the selection condition is EndDate < Date(). In summary, the implementation of the transaction is shown in Figure 5.10.

```
void Proj_End(PROJECT  proj[], int size_proj);
{ int i;
  struct PROJECT result[];
  int size_result;

 size_result = 0;
 for (i = 0; i < size_proj; i++)
   if (proj[i].EndDate < Date())
   {
       result[size_result] = proj[i];
       size_result++;
   }
 Print_relation(result, size_result);
}
```

Figure 5.10 Implementation of transaction 1 as a C function

Transaction 3 requests full details of all project assignments of male employees. It was specified using Z in section 5.2 and is again listed as follows:

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

All_Work

Ξ DB

wk! : P WORK

wk! = {wk : works, em : empls | wk.ENum = em.ENum $\wedge$ em.Sex = Male
• (wk.ENum, wk.PNum, wk.Hours }

The descriptor set of transaction 3, {<output, old>, <select, 1>, <join, 2>} is not exactly the same as that of either the minimum or the maximum requirements. Due to the difference between the descriptor sets, adaptation other than mere substitution needs to be performed. Table 5.8 indicates that the modification effort from the minimum requirements is less than that from the maximum requirements and suggests Cluster2_Min as the framework to be reused by the implementation of the transaction. As for transaction 1, the name of the schema is used as the name of the function and the function returns *void* since there is no output parameter. The input parameters as indicated in the framework are two relations involved in the *join* operation and their corresponding sizes. From the Z schema, we find that the two relations are *WORK* and *EMPLOYEE*. The descriptor set of the transaction indicates that there is a *select* operation to allow retrieval of only a subset of the contents in the database. The Z schema reveals that the relation involved in the selection is *EMPLOYEE*. Since the relation participates in both operations, as suggested by the adaptation process in Section 4.3.2, there is only one occurrence of the relation in the function prototype. The substitution of the resulting relation is similar to that of transaction 1. The body of the framework needs modification in addition to mere substitution of relation names. The select condition is to have the sex of the employees be "male". Therefore, the first search should be performed on relation *EMPLOYEE* and further search is only

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

necessary when the tuple satisfies the select condition. In summary, the transaction is implemented as a C function as shown in Figure 5.11.

```
void All_Work(EMPLOYEE  empls [], int size_empls,WORK  wk[], int size_wk);
      { int i, j;
        struct  WORK result[];
        int  size_result;
       size_result = 0;
       for (i = 0; i < size_empls; i++)
         if  (empls[i].sex < "Male")
             for (j = 0; j < size_wk; j++)
             {     result[size_result] = wk[j];
                  size_result++;            }
       Print_relation(result, size_result);
      }
```

Figure 5.11 Implementation of transaction 3 as a C function

Transaction 4 requests the SEX of all employees worked on project p1. It was specified using Z in Section 5.2 and is again listed as follows:

Sex_p

Ξ DB

lsex! : P SEX

lsex! = {em : empls; wk : works | wk.ENum = em.ENum $\wedge$ wk.PNum = p1 • (em.Sex)}

The descriptor set of transaction 4, {<output, new>, <select, 1>, <join, 2>, is not exactly the same as that of either the minimum or the maximum requirements of cluster 2. The difference between the descriptor sets implies that adaptation other than mere substitution needs to be performed. Table 5.8 indicates that the modification effort from the minimum requirements is greater than that from the maximum requirements and suggests Cluster2_Max as the framework to be reused by the implementation of the transaction. As for transaction 1, the name of the schema is used as the name of the function and the function returns *void* since there is no output parameter. The input parameters as indicated in the framework are: two relations involved in the *select*

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

operation and two relations involved in the *join* operation, and their corresponding sizes. The descriptor set of the transaction indicates that only one relation is involved in the *select* operation to allow retrieval of a subset of the contents in the database. The Z schema reveals that the relation involved in the selection is *WORK*. Since the relation participates in both *select* and *join* operations, as suggested by the adaptation process in section 4.3.2, there is only one occurrence of the relation in the function prototype. From the Z schema, we find out the other relation that is involved in the *join* operation is *EMPLOYEE*. Therefore, the function prototype for transaction 4 includes both relations and their corresponding sizes. The descriptor set of the transaction indicates that the resulting relation has to define a new structure to store the information retrieved. The "Resulting Relation" section of the framework provides the template for defining a new structure. The implementation of the transaction needs to fill in the template by a detailed list of fields according to the Z schema of the transaction. In addition, the body of the framework needs modification in addition to mere substitution of relation names. The framework contains a nested FOR loop to search through the two input relations simultaneously. In addition, both relations are applied by a filter to allow only the records that satisfy certain criteria man be included in the resulting relation. The implementation of the transaction needs to remove the second select condition and make sure that the remaining one applies to relation *WORK* as indicated in Z schema. Assigning retrieved data into the resulting relation requires field by field assignment. In summary, the transaction is implemented using C function as shown in Figure 5.12.

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

```
void Sex_Proj (EMPLOYEE  em[], int size_em, WORK  wk[], int size_wk);
   { int i, j;
     struct new_relation
          { SEX : sex; }
     new_relation  result[];
     int size_result;

     size_result = 0;
     for (i = 0; i < size_wk; i++)
       if (wk[i].PNum < "p1")
          for (j = 0; j < size_em; j++)
          if (wk[i].ENum = = em[j].ENum)
          {    result[size_result].sex = em[j].Sex;
               size_result++;    }
    Print_relation(result, size_result);
   }
```

Figure 5.12 Implementation of transaction 4 as a C function

The other twenty-three transactions in Section 5.2 are implemented according to a similar process as discussed for the first three transactions. Figures 5.13 to 5.34 list the resulting C implementation of the remaining twenty-three transactions.

5.5 Summary

In this chapter, a detailed case study was presented to demonstrate the effectiveness of RRM. The twenty five transactions reflect real-life features of a relational database application. The transactions are specified using Z. The extraction of the descriptor sets follows strict parsing rules. Clustering analysis revealed four clusters of similar transactions. Two frameworks for each cluster were designed, resulting in eight initial implementations. Instead of designing the twenty five schemas from scratch, the amount of effort to design from scratch is reduced to developing the eight frameworks, i.e. a reuse rate of 60%. As mentioned in Section 5.4, for clusters

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

whose sizes are small, as in this application, only one framework is necessary. In that case, the amount of effort to design from scratch is reduced to developing four frameworks, producing a reuse rate of approximately 85%. The case study has demonstrated that RRM can achieve a high reuse rate.

```
void Emp_Min(EMPLOYEE  empls[], int size_empls);
{ int i;
  struct EMPLOYEE result[];
  int size_result;

 size_result = 0;
 for (i = 0; i < size_empls; i++)
   if (empls[i].Rate = = 4.25)
   {
       result[size_result] = empls[i];
       size_result++;
   }
 Print_relation(result, size_result);
}
```

Figure 5.13 Implementation of transaction 2 as a C function

```
void Emp_Min(EMPLOYEE  empls[], int size_empls);
{ int i;
  struct EMPLOYEE result[];
  int size_result;

 size_result = 0;
 for (i = 0; i < size_empls; i++)
   if (empls[i].Sex = = "Male")
   {
       result[size_result] = empls[i];
       size_result++;
   }
 Print_relation(result, size_result);
}
```

Figure 5.14 Implementation of transaction 5 as a C function

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

```
void Fem_Mng (EMPLOYEE  empls[], int size_empls,
              DEPARTMENT  depts[], int size_depts);
{ int i, j;
  struct new_relation
       { ENUM : enum; }
  new_relation  result[];
  int  size_result;

 size_result = 0;
 for (i = 0; i < size_depts; i++)
   if  (depts[i].DCity < "New York")
       for (j = 0; j < size_empls; j++)
       if ((depts[i].ManENum = = em.ENum) && (em.Sex = = "Female"))
       {
            result[size_result].sex = empls[j].Sex;
            size_result++;
       }
 Print_relation(result, size_result);
}
```

Figure 5.15 Implementation of transaction 6 as a C function

```
void Id_Rate (EMPLOYEE  empls[], int size_empls,
              WORK  works[], int size_works);
{ int i, j;
  struct new_relation
       { ENUM : enum;
         RATE : rate;  }
  new_relation  result[];
  int  size_result;

 size_result = 0;
 for (i = 0; i < size_empls; i++)
   if  ((works[i].PNum < "p2") && (works[i].Hour > 100))
       for (j = 0; j < size_empls; j++)
       if (empls[j].ENum = = works[i].ENum)
       {    result[size_result].sex = empls[j].Sex;
            size_result++;    }
 Print_relation(result, size_result);
}
```

Figure 5.16 Implementation of transaction 7 as a C function

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

```
void Fem_Mng (EMPLOYEE  empls[], int size_empls,
              PROJECT  projs[], int size_projs,
              WORK works[], int size_works);
{ int i, j;
  struct new_relation
       { ENUM : enum; }
  new_relation  result[];
  int size_result;

  size_result = 0;
  for (i = 0; i < size_empls; i++)
    for (j = 0; j < size_projs; j++)
       if (empls[i]. HiredBy = = projs[j].CtrlBy)
          for (k = 0; k < size_works; k++)
             if  ((empls[i].ENum = = works[k].ENum) &&
                     (works[k].PNum = = projs[j].PNum))
                {    result[size_result].enum = empls[j].ENum;
                     size_result++;
                }
 Print_relation(result, size_result);
}
```

Figure 5.17 Implementation of transaction 8 as a C function

```
void Exp_Mng (EMPLOYEE  em[], int size_em, DEPARTMENT  dt[], int size_dt)
    { int i, j;
      struct new_relation
           { DNUM : dnum; }
      new_relation  result[];
      int size_result;

      size_result = 0;
      for (i = 0; i < size_dt; i++)
        if  (strcmp(dt[i].DCity, "New York"))
           for (j = 0; j < size_em; j++)
           if ((dt[i].ManENum = = em[j].ENum)
                  && (strcmp(em[j].Sex , "Female")))
           {    result[size_result].dnum = dt[i].DNum;
                size_result++;    }
     Print_relation(result, size_result);
    }
```

Figure 5.18Implementation of transaction 9 as a C function

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

```
void Male_Mng (EMPLOYEE  empls[], int size_empls,
                DEPARTMENT  depts[], int size_depts);
{ int i, j;
  struct DEPARTMENT  result[];
  int size_result;

 size_result = 0;
 for (i = 0; i < size_empls; i++)
   if (strcmp(empls[i].Sex , "Male"))
       for (j = 0; j < size_depts; j++)
       if (depts[i].ManENum = = empl[i].ENum)
       {    result[size_result] = depts[j];
            size_result++;    }
 Print_relation(result, size_result);
}
```

Figure 5.19 Implementation of transaction 10 as a C function

```
void Proj_Fem_Mng (EMPLOYEE  empls[], int size_empls,
                PROJECT  projs[], int size_projs,
                DEPARTMENT depts[], int size_depts);
{ int i, j, k;
  struct new_relation
       { PNUM : pnum; }
  new_relation  result[];
  int size_result;

 size_result = 0;
 for (i = 0; i < size_empls; i++)
   if (strcmp(empls[i].Sex, "Female"))
   for (j = 0; j < size_projs; j++)
       if (strcmp(projs[j].PCity, "New York"))
          for (k = 0; k < size_depts; k++)
             if ((empls[i].ENum = = depts[k].ManENum) &&
                    (depts[k].DNum = = projs[j].CtrlBy))
               {    result[size_result].pnum = projs[j].PNum;
                    size_result++;              }
 Print_relation(result, size_result);
}
```

Figure 5.20 Implementation of transaction 11 as a C function

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

```
void Low_Rate (EMPLOYEE  empls[], int size_empls);
{ int i;
  struct new_relation
       { ENUM enum;
         SEX sex; }
  new_relation  result[];
  int size_result;

 size_result = 0;
 for (i = 0; i < size_empls; i++)
    if ((empls[i].Rate < 10)
       {
            result[size_result].enum = empls[i].ENum;
            strcpy(result[size_result], empls[i].Sex);
            size_result++;
       }
 Print_relation(result, size_result);
}
```

Figure 5.21 Implementation of transaction 12 as a C function

```
void Proj_E (WORK  works[], int size_works);
{ int i;
  ENUM value;
  struct new_relation
       { PNUM pnum;
         HOUR hours; }
  new_relation  result[];
  int size_result;

 Get_value(&value);
 size_result = 0;
 for (i = 0; i < size_empls; i++)
    if (works[i].ENum = = value)
       {
            result[size_result].pnum = works[i].PNum;
            result[size_result].hours = works[i].Hour;
            size_result++;
       }
 Print_relation(result, size_result);
}
```

Figure 5.22 Implementation of transaction 13 as a C function

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

```
void Dept_City (DEPARTMENT  depts[], int size_depts);
{ int i;
  CITY value;
  struct new_relation
       { DNUM dnum; }
  new_relation  result[];
  int size_result;

 Get_value(&value);
 size_result = 0;
 for (i = 0; i < size_depts; i++)
    if (strcmp(depts[i].DCity, value))
       {
            result[size_result].dnum = depts[i].DNum;
            size_result++;
       }
 Print_relation(result, size_result);
}
```

Figure 5.23 Implementation of transaction 14 as a C function

```
void Proj_City (PROJECT  empls[], int size_empls,
                DEPARTMENT  depts[], int size_depts);
{ int i, j;
  struct PROJECT  result[];
  int size_result;

 size_result = 0;
 for (i = 0; i < size_projs; i++)
       for (j = 0; j < size_depts; j++)
       if ((projs[i].CtrlBy = = depts[j].DNum) &&
            (strcmp(projs[i].PCity, depts[j].DCity)))
       {
            result[size_result] = projs[i];
            size_result++;
       }
 Print_relation(result, size_result);
}
```

Figure 5.24 Implementation of transaction 16 as a C function

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

```
void Emp_Hired_P (EMPLOYEE  empls[], int size_empls,
                  PROJECT  projs[], int size_projs,
                  WORK works[], int size_works);
{ int i, j, k;
  PNUM value;
  struct new_relation
       { ENUM enum;
         RATE rate}
  new_relation  result[];
  int size_result;

  Get_value(&value);
  size_result = 0;
  for (i = 0; i < size_projs; i++)
    if (strcmp(projs[i].PCity, value))
    for (j = 0; j < size_works; j++)
        if (projs[i].PNum = = works[j].PNum)
           for (k = 0; k < size_empls; k++)
              if ((projs[i].StartDate < empls[k].HiredDate) &&
                        (empls[k].ENum = = works[j].ENum))
                 {    result[size_result].enum = empls[k].ENum;
                      result[size_result].rate = empls[k].Rate;
                      size_result++;              }
 Print_relation(result, size_result);
}
```

Figure 5.25 Implementation of transaction 15 as a C function

```
void Dept_NY(DEPARTMENT  depts[], int size_depts);
{ int i;
  struct DEPARTMENT result[];
  int size_result;

  size_result = 0;
  for (i = 0; i < size_depts; i++)
    if (strcmp(depts[i].DCity, "New York"))
    {   result[size_result] = depts[i];
        size_result++;
    }
 Print_relation(result, size_result);
}
```

Figure 5.26 Implementation of transaction 18 as a C function

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

```
void P_Ctrl_D (EMPLOYEE  empls[], int size_empls,
                         PROJECT  projs[], int size_projs,
                         DEPARTMENT depts[], int size_depts);
{ int i, j, k;
  DNUM value;
  struct PROJECT result[];
  int size_result;

 Get_value(&value);
 size_result = 0;
 for (i = 0; i < size_projs; i++)
   if (projs[i].CtrlBy = = value)
   for (j = 0; j < size_depts; j++)
       if (projs[i].CtrlBy = = depts[j].DNum)
          for (k = 0; k < size_empls; k++)
             if ((projs[i].EndDate < empls[k].HiredDate) &&
                         (empls[k].ENum = = depts[j].ManENum))
                {    result[size_result] = projs[i];
                     size_result++;            }
 Print_relation(result, size_result);
}
```

Figure 5.27 Implementation of transaction 17 as a C function

```
void Proj_Ctrl_D(PROJECT  projs[], int size_projs);
{ int i;
  DNUM value;
  struct PROJECT result[];
  int size_result;

 Get_value(&value);
 size_result = 0;
 for (i = 0; i < size_projs; i++)
   if (projs[i].CtrlBy = = value)
   {
       result[size_result] = empls[i];
       size_result++;
   }
 Print_relation(result, size_result);
}
```

Figure 5.28 Implementation of transaction 19 as a C function

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

```
void Emp_PJ_NY (EMPLOYEE  empls[], int size_empls,
                PROJECT  projs[], int size_projs,
                WORK works[], int size_works);
{ int i, j, k;
  struct new_relation
       { ENUM enum;
         RATE rate }
  new_relation  result[];
  int size_result;

  size_result = 0;
  for (i = 0; i < size_projs; i++)
    if (strcmp(projs[i].PCity, "New York"))
    for (j = 0; j < size_empls; j++)
        if (strcmp(empls[j].Sex, "Female"))
           for (k = 0; k < size_works; k++)
              if  ((projs[i].PNum = = works[k].PNum) &&
                        (empls[j].ENum = = works[k].ENum))
                 {    result[size_result].enum = empls[k].ENum;
                      result[size_result].rate = empls[k].Rate;
                      size_result++;              }
 Print_relation(result, size_result);
}
```

Figure 5.29 Implementation of transaction 20 as a C function

```
void Emp_Date(EMPLOYEE  empls[], int size_empls);
{ int i;
  DATE value1, value2;
  struct EMPLOYEE result[];
  int size_result;

  Get_value(&value1, &value2);
  size_result = 0;
  for (i = 0; i < size_empls; i++)
    if  ((empls[i].HiredDate >= value1) && (empls[i].HiredDate <= value2))
    {   result[size_result] = empls[i];
        size_result++;     }
 Print_relation(result, size_result);
}
```

Figure 5.30 Implementation of transaction 21 as a C function

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

```
void Emp_Hour(WORKS   works[], int size_works);
{  int  i;
   PNUM value;
   struct  new_relation
         { ENUM  enum; }
   new_relation  result[];
   int  size_result;

  Get_value(&value);
  size_result = 0;
  for (i = 0; i < size_works; i++)
    if  (works[i].PNum = = value)
    {   result[size_result].enum = works[i].ENum;
        size_result++;     }
 Print_relation(result, size_result);
}
```

Figure 5.31 Implementation of transaction 22 as a C function

```
void Dept_PJ_NY (DEPARTMENT   depts[], int size_depts,
                 PROJECT  projs[], int size_projs,
                 EMPLOYEE empls[], int size_empls);
{  int  i, j, k;
   struct  new_relation
         { DNUM  enum;
           RATE rate }
   new_relation  result[];
   int  size_result;

  size_result = 0;
  for (i = 0; i < size_projs; i++)
    if (strcmp(projs[i].PCity, "New York"))
    for (j = 0; j < size_depts; j++)
        if (projs[i].CtrlBy = = depts[j].DNum)
           for (k = 0; k < size_empls; k++)
              if  ((depts[j].ManENum = = empls[k].ENum)
                 {     result[size_result].dnum = depts[j].DNum;
                       result[size_result].rate = empls[k].Rate;
                       size_result++;              }
 Print_relation(result, size_result);
}
```

Figure 5.32 Implementation of transaction 23 as a C function

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

```
void Proj_Fem (EMPLOYEE  empls[], int size_empls,
               WORK  works[], int size_works);
{ int i, j;
  struct new_relation
       { ENUM enum;
         HOUR hours}
  new_relation  result[];
  int size_result;

  size_result = 0;
  for (i = 0; i < size_empls; i++)
    if (strcmp(empls[i].Sex, "Female"))
       for (j = 0; j < size_works; j++)
       if (empls[i].ENum = = works[j].ENum)
       {     result[size_result].enum = empls[i].ENum;
             result[size_result].hours = works[j].Hour;
             size_result++;    }
 Print_relation(result, size_result);
}
```

Figure 5.33 Implementation of transaction 24 as a C function

```
void Rate_Male(EMPLOYEE  empls[], int size_empls);
{ int i;
  struct new_relation
       { ENUM enum;
         RATE rate}
  new_relation  result[];
  int size_result;

  size_result = 0;
  for (i = 0; i < size_empls; i++)
    if (strcmp(empls[i].Sex, "Male"))
    {
       result[size_result].enum = empls[i].ENum;
       result[size_result].rate = empls[i].Rate;
       size_result++;
    }
 Print_relation(result, size_result);
}
```

Figure 5.34 Implementation of transaction 25 as a C function

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

Chapter 6

Summary

Although there have been intensive research attempts and industrial projects in software reuse for two decades, limited progress has been made to implement reuse in practice. The purpose of this research was to develop a reuse model that integrates software reuse as a part of the software development lifecycle. The third and fourth chapters presented the two major phases of the Requirement Reusability Model(RRM).

The first phase of RRM is the Reusability Analysis (RA) phase which includes a clustering structure so that transactions in the same cluster share more functionality than transactions in different clusters. The formal specification language Z was used to represent the functional semantics of transactions. A parsing process was proposed to automatically detect the functionality of each transaction. The results of the parsing process included a data dictionary and the first version of the transaction dictionary. Based on information in the two dictionaries, algorithms were defined to compute the similarity indices among the transactions. The similarity between any two given transactions measures the percentage of common functionality of the two transactions. The cluster hierarchy was generated by a modified version of the HAC method where the process stops when the similarity indices of any two intermediate clusters fall below a threshold value. The similarity index of two clusters is computed based on the similarity indices among transactions in the clusters.

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

The second phase of RRM is called Reusability Realization (RR) phase which creates and recommends reusable frameworks for reuse by implementation of individual transactions. Generic requirements for the clusters inferred from the RA phase were extracted to identify the common functionality requested by the transactions that belong to the same cluster. Frameworks were generated to accomplish the generic requirements. There were three major parts of a framework: the interface, the local variables, and the body of the framework. For any given transaction, the recommendation of the framework to be reused was done through the estimation of adaptation efforts from the framework to the implementation of the transaction. Guidelines were provided for adaptation of the recommended framework to implement the transaction. A similar process also applies to the situation when a new transaction is requested to extend the application.

A case study was presented in Chapter 5 to demonstrate the effectiveness of RRM. In this chapter, the contributions of RRM are discussed in Section 6.1 and the future research is proposed in Section 6.2.

6.1 Research Contributions

Current reuse methodologies require a repository of components to be reused by the implementation of a software system. The reuse research community treats the repository similar to a library of books. Therefore, the first challenge of repository-based software reuse comes from selecting and/or building appropriate components to be included in the repository. Clearly, a library cannot contain every book. In addition, it is not efficient to keep books that will never be read by the users of the library. By the same analogy, only the components that have the potential to be reused should be

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

included in the reuse repository. However, most efforts in software reuse have been invested in the organization of the reuse repositories, that is how to represent, store, and retrieve components. The assumption is that the components to be included in the repository are readily available. There has been no objective basis for deciding whether a component should be included in such repositories. This research provides a solution to this problem by performing reusability analysis on the requirement specifications. It identifies groups of transactions in an application so that transactions in the same groups share more common functionality than transactions in the different groups. A component is included in the repository only if it possesses common functionality for a group of transactions. This research represents a unique approach to guarantee that the components in the repository are useful for reuse purposes. It is significant because it provides a formal method to determine what components should be placed in the repository.

The second challenge of repository-based software reuse comes from organizing the repository so that it is easy to find the best candidate component to be reused for the implementation of the transactions. The success of the repository depends on the representation of the components and the organization of the components. The organization scheme in RRM is a combination of search by profile and match through verification in that a formal specification is used to describe each transaction and the extraction of profiles is based on formal specifications instead of text documents. The formal specification is a mathematical documentation; therefore, the functional semantics of the components is used rather than the conceptual semantics. In addition, using a mathematics based formal specification overcomes the

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

ambiguity of natural languages. On the other hand, current specification-based component retrieval methods match query specification to specifications in the library through theorem proving which is very expensive. RRM employs clustering analysis to group transactions into homogeneous clusters. Since the new organization scheme and clustering method require significantly less human effort than does theorem proving, efficiency for the reuse process is improved by the application of RRM.

The third challenge of repository-based software reuse comes from the fact that we retrieve components for use in the new application so that we do not have to write the implementation from scratch. This retrieval is not the same as retrieving a book from a library where the job is complete once the book is found. It is rarely the case that a component retrieved serves the exact same functionality under the exact same environment. Modification is typically necessary. Most current repositories do not provide guidelines for adaptation. RRM deals with the problem by ensuring that the frameworks in the repository are represented in the same way as the transactions that are to be implemented. A comparison between two entities that are described under the same measurement space is much easier than a comparison between two entities that are described by different methods. In addition, the frameworks serve as the templates for the implementation of individual transactions and guidelines are provided to assist the adaptation process. Thus, this research provides an innovative representation scheme that reduces the amount of effort required to adapt the retrieved component.

Another contribution of this research is that it represents a unique way to integrate software reuse as a part of the software lifecycle. Most research in software reuse deals with the cross-system reuse which requires that a repository of reusable

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

components is available when a new software system is initiated. Little attention has been paid to the repetitive works that exist in the development of a single system. RRM makes no assumption of an existing repository but rather infers a repository of reusable frameworks. The two phases of RRM can be regarded as the additional phases of the software lifecycle between the requirement specification analysis phase and the implementation/adaptation phase. The formal specification of any given application, the result of the requirement specification analysis phase, serves as the input to RRM. RRM generates a repository of frameworks and the mapping between the frameworks and the transactions. The implementation/adaptation phase, then, carries out the modifications that are suggested by RRM. The result is that this research defines a new software lifecycle that integrates reuse in a systematic manner.

Formal-based approaches and solutions are often viewed as too complicated and, hence, not accepted by industrial software engineers. The consequence is that software development suffers fundamentally from this lack of formality. Success in introducing formal methods into industry depends on making those methods cost-effective in the overall context of industrial software engineering practice. First, the potential of reusability is identified right after the specification phase so that unnecessary duplication of effort in implementation is avoided. Secondly, the only reusable frameworks generated by RRM are those that are useful for implementation of a group of transactions. Finally, by assigning the same cluster of transactions to the same group of engineers for implementation, time required to change from one unrelated activity to the next is reduced. RRM advances the use of formal

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

specifications by demonstrating that the investment required to write formal specification can be amortized later in the development.

Finally, RRM represents a model that uniquely aids the evolution of a software system by providing a systematic way to expand the software systems. When a new transaction is requested, RRM analyzes the specification of the transaction and recommends a framework from the repository that is most similar to the functionality of the transaction requested. The implementation of the transaction does not have to start from scratch but rather can reuse the framework recommended. Thus this method uniquely extends the reuse potential to the software maintenance phase.

6.2 Future Research

A prototype system, REST, was implemented to demonstrate the effectiveness of RRM. It can be extended as a complete CASE tool with user-friendly interface. The first component of the CASE tool would be a specially designed pattern recognizer which takes the formal specification of an application as its input and identifies the basic operations requested by individual transactions. The second component of the CASE tool would internally generate cluster structure. The third component of the CASE tool would provide a graphical user interface that allow users to simultaneously access information such as the frameworks recommended, the descriptor sets of the transactions to be implemented and the frameworks to be reused, and the guidelines provided for adaptation process.

RRM uses a threshold value to determine the final clusters of transactions. The clustering analysis terminates when the minimum similarity among transactions of the same cluster falls below the threshold value. An extension to RRM could keep the

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

complete cluster hierarchy along with the minimum similarity preserved by each intermediate cluster. Frameworks can be generated for all intermediate clusters. At the time of implementation, the user would be given options of a similarity interval. A search would be conducted on the cluster hierarchy to determine which cluster contains the given transaction and preserves a similarity that is within the similarity interval provided by the users. The framework for the cluster found would be recommended for reuse.

Another extension to this research is to apply the process proposed in RRM to other application domains. The major phases of RRM remain the same. Modifications need to be made to the patterns that are used to identify the basic operations of the new application domain. Domain modeling techniques can be combined with RRM to automate the process of identifying common patterns.

Object technology is believed to be crucial to achieving the promise of software reuse. However, using objects does not automatically ensure reuse. In fact, in has been shown that the concepts of inheritance and polymorphism introduce severe difficulties in maintaining and reusing programs [Lej 92, Wil 92]. To obtain a true systematic object-oriented reuse process, current object technology must be augmented with specific reuse-oriented processes, organization structures, guidelines, and training [Gri 95]. RRM can be extended to help identify optimal class hierarchies within an object-oriented application at the design level.

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

References

[Ade 90] Ader, M., Nierstrasz, O., McMahon, S., Muller, G. and Profrock, A-K "The Ithaca Technology: a Landscape for Object-Oriented Application Development" In *Proceedings of ESPRIT'90* Kluwer Academic Publisher, November 1990

[Agr 87] Agresti, W. W. *New Paradigms for Software Development* IEEE Computer Society Press, 1987

[Bar 94] Barros, R. S. M. *On the Formal Specification and Derivation of Relational Database Applications* Ph.D. Dissertation, University of Glasgow, UK November 1994

[Big 87] Biggerstaff, T. J. and Richter, C. "Reusability Framework, Assessment, and Directions" *IEEE Software* Vol. 4, No. 2, pp 41 - 49 March 1987

[Big 94] Biggerstaff, T. "The Library Scaling Problem and the Limits of Concrete Component Reuse" In *Proceedings of 3rd International Conference on Software Reusability* pp 190 - 197 Rio de Janeiro, Brazil, IEEE Computer Society Press 1994

[Boe 87] Boehm, B. "Improving Software Productivity" *IEEE Software* Vol. 4, No. 4, pp 43 - 57 July 1987

[Boe 88] Boehm, B. "A Spiral Model of Software Development and Enhancement" *IEEE Computer* Vol. 21, No. 5, pp 61 - 72 May 1988

[Bro 87] Brooks, F. "No Silver Bullet: Essence and Accidents of Software Engineering" *IEEE Computer* Vol. 20, No. 4, pp 10 - 19 April 1987

[Cas 93] Castano, S. and De Antonellis, V. "A Constructive Approach to Reuse of Conceptual Components" In *Proceedings of the 2nd International Workshop on Software Reusability* pp 19 - 28 Lucca, Italy March 1993

[Cas 94] Castano, S. and De Antonellis, V "The F^3 Reuse Environment for Requirements Engineering" *ACM SIGSOFT Software Engineering Notes* Vol. 19, No. 3, July 1994

[Cas 95] Castano, S., De Antonellis, V. and Pernici, B. "Building Reusable Components in the Public Administration Domain" In *Proceedings of the Symposium on Software Reusability* pp 81 - 87 Seattle, WA April 1995

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

[Che 76] Chen, P. P. "The Entity-Relationship Model -- Toward a Unified View of Data" *ACM Transactions on Database Systems* Vol. 1, No. 1, pp 9 - 36 January 1976

[Che 93] Chen, P. S., Hennicker, R. and Jarke, M. "On the Retrieval of Reusable Software Components" In *Proceedings of the 2nd International Workshop on Software Reusability* pp 99 - 108 Lucca, Italy March 1993

[Dat 90] Date, C. J. *An Introduction to Database Systems* Vol. 1, World Student Series, Addison-Wesley Publishing Company Inc., Reading, Massachusetts, USA, 5th ed., 1990

[Dev 90] Devanbu, P. R., Brachman, R. J. and Selfridge, P. G. "LaSSIE: A Classification-based Software Information System" In *Proceedings of the 12th International Conference on Software Engineering* March 1990

[Dun 93] Dunn, M. F. and Knight, J. C. "Automating the Detection of Reusable Parts in Existing Software" In *Proceedings of the 15th International Conference on Software Engineering* pp 381 - 390 Baltimore, MD 1993

[Emb 87] Embley, D. W. and Woodfield, S. N. "A Knowledge Structure for Reusing Abstract Data Types" In *Proceedings of the 9th International Conference on Software Engineering* pp 360 - 368 1987

[Est 91] Esteva, J. C. and Reynolds, R. G. "Identifying Reusable Software Components by Induction" *International Journal of Software Engineering and Knowledge Engineering* Vol. 1, No. 3 pp 271 - 292 March 1991

[Est 95] Esteva, J. C. "Automatic Identification of Reusable Components" In *Proceedings of the 7th International Workshop on Computer-Aided Software Engineering* pp 80 - 87 1995

[Fei 92] Feiler, P., and Humphrey, W. "Software Process Development and Enactment: Concepts and Definitions" *CMU/SEI-92-TR-04* Software Engineering Institute, Pittsburgh, PA, September 1992

[Fic 88] Fickas, S. and Nagarajan, P. "Critiquing Software Specifications" *IEEE Software* Vol. 5, No. 6 November 1988

[Fie 88] Field, A. J. and Harrison P. G. *Functional Programming* Addison-Wesley, 1988

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

[Fug 92] Fugini, M. G. and Pernici, B. "Reusable Requirements" In *Conceptual Modeling, Databases and CASE* P. Loucopoulos and R. Zicari Eds. J. Wiley & Sons, 1992

[Fug 93] Fugini, M. G. and Faustle, S. "Retrieval of Reusable Components in a Development Information System" In *Proceedings of the 2nd International Workshop in Software Reusability* pp 89 - 98 Lucca, Italy March 24 - 26, 1993

[Gal 95] Gall, H., Jazayeri, M. and Klosch, R. "Future Research Directions in SW-Reuse: Where to Go from Here" In *Proceedings of the Symposium on Software Reusability* pp 225 - 228 Seattle, WA April 1995

[Gar 90] Garlan, D. "The Role of Formal Reusable Frameworks" *Proceedings of the ACM SIGSOFT International Workshop on Formal Methods in Software Development* pp 42 - 44, Napa, CA, May 9 - 11, 1990

[Gar 95] Garlan, D. "Research Directions in Software Architecture" *ACM Computing Surveys* Vol. 27, No. 2, pp 257 - 261 June 1995

[Gri 95] Griss, M., Jacobson, I., Jette, C., Kessler, B. and Lea, D. "Panel: Systematic Software Reuse Objects and Frameworks are not Enough" In *Proceedings of the Symposium on Software Reusability* pp 17 - 20 Seattle, WA, April 1995

[Hal 90] Hall, A. "Seven Myths of Formal Methods" *IEEE Software* Vol. 7, No. 5 September 1990

[Hol 96] Hollenbach, C. and Frakes, W. *Software Process Reuse in an Industrial Setting* Technical Report, TR-96-04 Virginia Polytechnic Institute and State University, January 1996

[Hor 89] Horowitz, E. and Munson, J. B. "An Expansive View of Reusable Software" In *Software Reusability: Concepts and Models* pp 19 - 41 Biggerstaff, T. J. and Perlis, A. J. eds. ACM Press, New York 1989

[Isa 96] Isakowitz, T. and Kauffman, R. J. "Supporting Search for Reusable Software Objects" *IEEE Transactions on Software Engineering* Vol. 22, No. 6, pp 407 - 423 June 1996

[Joh 92] Johnson, W. L., Feather, M. and Harris, D. "Representation and Presentation of Requirements knowledge" *IEEE Trans. on Software Engineering* Vol. 18, No. 10 October 1992

[Joh 95] Johnson, R. E. "Why Doesn't the Reuse Community Talk About Reusable Software" In *Proceedings of the 7th WISR* 1995

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

[Jon 84] Jones, T. C. "Reusability in Programming: A Survey of the State of the Art" *IEEE Transactions on Software Engineering* Vol. 10, No. 5 pp 488 - 494 May 1984

[Jon 88] Jones, G. and Prieto-Diaz, R. "Building and Managing Software Libraries" In *Proceedings of the 12th Annual International Computer Software and Applications Conference* pp 228 - 236 October 1988

[Kat 87] Katz, S., Richter, C. A. and The, K. "PARIS: A System for Reusing Partially Interpreted Schemas" In *Proceedings of the 9th International Conference on Software Engineering* pp 377 - 385 1987

[Kru 92] Krueger, C. W. "Software Reuse" *ACM Computing Surveys* Vol. 24, No. 2, pp 131 - 183 March 1992

[Ku 93] Ku, B. K. "SRSGEN - A Software Reuse Tool For Large Scale Applications" In *Proceedings of the 2nd International Workshop in Software Reusability* pp 109 - 115 Lucca, Italy March 24 - 26, 1993

[Lan 84] Lanergan, R. G. and Grasso, C. A. "Software Engineering with Reusable Design and Code" *IEEE Transactions on Software Engineering*, Vol. 10, No. 5 pp 498 - 501 September 1984

[Lat 95] Latour, L and Dusink, L. "Functional Fixedness in the Design of Software Artifacts" In *Proceedings of the 7th WISR* 1995

[Lej 92] Lejter, M. and Meyers, S. and Reiss, S. P. "Support for Maintaining Object-Oriented Programs" *IEEE Transactions on Software Engineering* Vol. 18, No. 12 pp 1045 - 1052 December 1992

[Maa 91] Maarek, Y. S., Berry, D. M. and Kaiser, G. E. "An Information Retrieval Approach for Automatically Constructing Software Libraries" *IEEE Transactions on Software Engineering* Vol. 17, No. 8, pp 800 - 813 August 1991

[May 95] Mayr, H. C. "Conceptual Predesign: A Platform for the Reuse of Requirements Specifications" In *Proceedings of the 7th WISR* 1995

[McI 68] McIlroy, M. D. "Mass Produced Software Components" In *Software Engineering: Report on a Conference by the NATO Science Committee* pp 88 - 98 Naur, P. and Randell, B. Eds NATO Scientific Affairs Division, Brussels, Belgium October 1968

[Mil 94] Mili, A., Mili, R. and Mittermeir, R. "Storing and Retrieving Software Components: A Refinement-based Approach" In *Proceedings of the*

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

16th International Conference on Software Engineering pp 91 - 100 1989

[Mil 95] Mili, H., Mili, F. and Mili, A. "Reusing Software: Issues and Research Directions" *IEEE Transactions on Software Engineering*, Vol. 21, No. 6 pp 528 - 561 June 1995

[Mir 91] Miriyala, K. and Harandi, M. T. "Automatic Derivation of Formal Software Specifications from Informal Descriptions" *IEEE Trans. on Software Engineering* Vol. 17, No. 10 pp October 1991

[Ost 92] Ostertag, E., Hendler, J., Prieto-Diaz, R. and Braun, C. "Computing Similarity in a Reuse Library System: An AI-Based Approach" *ACM Transactions on Software Engineering and Methodology* Vol. 1, No. 3, pp 205 - 228 July 1992

[Per 89] Perry, D. E. "The Inscape Environment" In *Proceedings of the 11th International Conference on Software Engineering* pp 2 - 12 1989

[Pil 96] Pillai, K. "The Fountain Model and Its Impact on Project Schedule" *ACM SIGSOFT Software Engineering Notes* Vol. 21, No. 2, pp 32 - 38 March 1996

[Pou 93] Poulin, J. S. and Yglesias, K. P. "Experiences with a Faceted Classification Scheme in a Large Reusable Software Library(RSL)" In *Proceedings of the 7th Annual International Computer Software and Applications Conference* pp 90 - 99 Phoenix, AZ, November 1993

[Pre 92] Pressman, R. S. *Software Engineering: A Practitioner's Approach* 3rd ed. McGraw-Hill, Inc. 1992

[Pri 87] Prieto-Diaz, R. and Freeman, P. "Classifying Software for Reusability" *IEEE Software* Vol. 4, No. 1, pp 6 - 16 January 1987

[Pri 91] Prieto-Diaz, R. "Implementing Faceted Classification for Software Reuse" *Communication of the ACM* Vol. 34, No. 5, pp 88 - 97 May 1991

[Pri 93] Prieto-Diaz, R. "Status Report: Software Reusability" *IEEE Software* Vol. 10, No. 3, pp 61 - 66 May 1993

[Pri 96] Prieto-Diaz, R. "Reuse as a New Paradigm for Software Development" In *System Reuse: Issues in Initiating and Improving a Reuse Program*, M. Sarshar ed. London : Springer-Verlag 1996

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

[Ran 57] Ranghanathan, S. R. *Prolegomena to Library Classification* The Garden City Press Ltd., Letchworth, Herfordshire, 1957

[Reu 91] Reubenstein, H. B. and Waters, R. C. "The Requirements Apprentice: Automated Assistance for Requirements Acquisition" *IEEE Trans. on Software Engineering* Vol. 17, No. 3 March 1991

[Rol 91] Rollins, E. J. and Wing, J. M. "Specifications as Search Keys for Software Libraries" In *Proceedings of the 8th International Conference on Logic Programming* June 1991

[Sit 95] Sitaraman, M., Fleming, D., Hopkins, J. and Sreerama, S. "Why (Not) Reuse (Typical) Code Components?" In *Proceedings of the 7th WISR* 1995

[Sol 84] Soloway, E. and Ehrlich, K. "Empirical Studies of Programming Knowledge" *IEEE Transactions on Software Engineering* Vol. 10, No. 5 pp 595 - 609 September 1984

[Sor 93] Sorumgard, L. S., Sindre, G. and Stokke, F. "Experiences from Application of a Faceted Classification Scheme" In *Proceedings of the 3rd International Workshop in Software Reusability* pp 116 - 124 Lucca, Italy March 24 - 26, 1993

[Spi 92] Spivey, J. M. *The Z Notation: A Reference Manual* Prentice Hall International(UK) Ltd. Hemel Hempstead, UK, 2nd ed. 1992

[Tra 90] Tracz, W. "Where Does Reuse Start?" *ACM SIGSOFT Software Engineering Notes* Vol. 15, No. 2, pp 42 - 46 April 1990

[Tur 87] Turner, J. A. "Understanding the Elements of System Design" In *Critical Issues in Information System Research*, R. J. Boland, Jr., and R. A. Hirschheim Ed. John Wiley, Chichester 1987

[Wem 92] Wemmerlov, U. and Hyer, N. *Handbook of Industrial Engineering* John Wiley & Sons 1992

[Wil 92] Wilde, N. and Huitt, R. "Maintenance Support for Object-Oriented Programs" *IEEE Transactions on Software Engineering* Vol. 18, No. 12 pp 1038 - 1044 December 1992

[Win 90] Wing, J. M. "A Specifier's Introduction to Formal Methods" *IEEE Computer* pp 9 - 24 September 1990

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

[Wor 92] Wordsworth, J. B. *Software Development with Z: A Practical Approach to Formal Methods in Software Engineering* International Computer Science Series, Addison-Wesley 1992

[Zar 95a] Zaremski, A. M. and Wing, J. M. "Signature Matching: a Tool for Using Software Libraries" *ACM Transactions on Software Engineering Methodology* Vol. 4, No. 2, pp 146 - 170 April 1995

[Zar 95b] Zaremski, A. M. and Wing, J. M. "Specification Matching of Software Components" In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundation of Software Engineering* pp 6 - 17 October 1995

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

Vita

Youwen Ouyang was born in Fuzhou, Fujian, People's Republic of China on June 16, 1966. She completed the Bachelor degree in Computer Science at Peking (Beijing) University, Beijing, People's Republic of China in July 1986. She received a Master of Science degree in Computer Science from Xiamen University, Xiamen, Fujian, People's Republic China in 1989. Since September 1990, she joined the Department of Computer Science in the doctoral program. She has been working as a teaching assistant since January 1991. In May 1994, she received a Master of Applied Statistics from Louisiana State University.

Her research interests include formal specification, software reusability, software re-engineering, object-oriented paradigm, database, and statistical applications in computer science.

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

DOCTORAL EXAMINATION AND DISSERTATION REPORT

Candidate: Youwen Ouyang

Major Field: Computer Science

Title of Dissertation: A Reusability Model That Creates Design Frameworks Using a Formal Specification Clustering Approach

Approved:

Major Professor and Chairman

Dean of the Graduate School

EXAMINING COMMITTEE:

Date of Examination:

August 27, 1996

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