MERGING DATA FROM MULTIPLE MANUFACTURING SOFTWARE SYSTEMS

A Thesis Presented to
The Faculty of the
Fritz J. and Dolores H. Russ College of Engineering and Technology
Ohio University
In Partial Fulfillment
of the Requirement for the Degree
Master of Science

by
Yizhong Wang
August, 1999
Acknowledgement

I am especially grateful to my adviser -- Dr. Robert Judd for introducing me to the computer software world. His patience and intelligence has made me a better programmer. As many people know, he solves problems before anybody else has even realized there are problems.

Mr. Dinesh Dhamija helped me in many significant ways in our research work. His insights into newly emerging concepts and comprehension of unfamiliar knowledge helped me to understand the complicated software logic.

Last, but not least, I must thank Dr. Dennis Irwin for all the financial support he provided when I was really thirsty for this chance to become more valuable to human society.
# Table of Contents

Acknowledgement........................................................................................................................................................................ iii

Table of Contents............................................................................................................................................................................... iv

List of Figures................................................................................................................................................................................... viii

Abbreviations.................................................................................................................................................................................... xii

Chapter 1 Introduction........................................................................................................................................................................ 1

1.1. Introduction ................................................................................................................................................................................ 1

1.2. Four Basic Problems of Integration ........................................................................................................................................ 2

1.3. Literature Review.......................................................................................................................................................................... 3

1.4. The IMDE Architecture .............................................................................................................................................................. 5

1.4.1. Unified Data Meta-Model ........................................................................................................................................................ 5

1.4.2. Intelligent Interface ................................................................................................................................................................ 6

1.5. Thesis Scope................................................................................................................................................................................. 7

1.6. Thesis Organization ................................................................................................................................................................... 7

Chapter 2 Internal Merge ................................................................................................................................................................. 9

2.1. Introduction ................................................................................................................................................................................ 9

2.2. Architecture ............................................................................................................................................................................... 10

2.3. Merge Rules ................................................................................................................................................................................. 11

2.4. Overall Approach ..................................................................................................................................................................... 15

2.5. Internal Merge Algorithms ....................................................................................................................................................... 17

2.5.1. Merge the Databases ......................................................................................................................................................... 17

2.5.1.1. Function MergeDatabase .............................................................................................................................................. 17

2.5.1.2. Function MergeAllEntities ............................................................................................................................................ 18

2.5.1.3. Function GetTree ............................................................................................................................................................. 19

2.5.2. Merge Each Entity Instance ............................................................................................................................................. 20

2.5.2.1. Function MergeEntityInstance ..................................................................................................................................... 20

2.5.2.1.1. Parameters ................................................................................................................................................................ 20

2.5.2.1.2. Description ................................................................................................................................................................. 22
2.5.6. Merge Relations *(CompleteRelation)* ............................................................... 48

2.5.6.1. Parameters ........................................................................................................ 48

2.5.6.2. Description ........................................................................................................ 48

2.6. Target Database Information .................................................................................. 49

2.7. Functions for Target Database Manipulation ......................................................... 52

2.7.1. Manipulate the Target .......................................................................................... 52

2.7.1.1. Function *dbDelete* .......................................................................................... 52

2.7.1.2. Function *dbShift* .......................................................................................... 54

2.7.2. Change References or Attributes ....................................................................... 55

2.7.2.1. Function *dbUpdateReferences* ..................................................................... 55

2.7.2.2. Function *dbCopyAttribs* ................................................................................. 57

2.7.3. Other Functions .................................................................................................. 58

2.7.3.1. Function *dbGetRootName* and *dbGetRootDef* ............................................ 58

2.7.3.2. Function *dbFindChild* .................................................................................. 59

2.7.3.3. Function *GetOIDs* ......................................................................................... 60

Chapter 3 Generic Examples of the Internal Merge ...................................................... 62

3.1. Introduction ............................................................................................................. 62

3.2. Merge Inheriting Entities ....................................................................................... 62

3.3. Check and Create Dependent Relations ................................................................. 66

3.4. Merge Aggregate Values ....................................................................................... 70

3.5. Merge Pointer Attributes ...................................................................................... 73

Chapter 4 Cost Advantage ............................................................................................. 78

4.1. Introduction ............................................................................................................. 78

4.2. CA Overview .......................................................................................................... 78
4.3. Communication with CA

4.3.1. Extract Data from CA—Report File

4.3.2. Update the Local Database of CA—Design File

4.4. Design of Two Translators

4.4.1. CA Data Schema

4.4.2. Translator for Extracting Data from CA

4.4.2.1. Function ReadFile

4.4.2.2. Function ReadComponent

4.4.2.3. Function ReadAssembly

4.4.3. Translator for Inserting Data into CA

4.4.3.1. Function InsertComponent

4.4.3.2. Function InsertAssembly

4.5. An Example of CA Merge

Chapter 5 Conclusions and Future Work

5.1. Review of Objectives and Accomplishments

5.2. Future Work

Bibliography
List of Figures

Chapter 2 Internal Merge

Figure 2-1 Internal Merge Architecture .......................................................... 10
Figure 2-2 EXPRESS Schema of Merge Rules .................................................. 13
Figure 2-3 Overall Approach ........................................................................... 16
Figure 2-4 Algorithm of Function MergeDatabase ........................................... 17
Figure 2-5 Algorithm of Function MergeAllEntities ........................................ 18
Figure 2-6 Algorithm of Function GetTree ....................................................... 19
Figure 2-7 Algorithm of Function MergeEntityInstance ................................. 21
Figure 2-8 Procedures for the OID Map ........................................................... 24
Figure 2-9 an Example of OID Map ................................................................. 24
Figure 2-10 Algorithm of Function FillInheritanceTree .................................. 25
Figure 2-11 Algorithm of Function Find .......................................................... 27
Figure 2-12 Algorithm of Function CheckDependencies ................................. 28
Figure 2-13 Algorithm of Function CreateEntity ............................................ 31
Figure 2-14 Algorithm of Function CreateAggreEntity .................................. 32
Figure 2-15 Algorithm of Function MergeInfoAttrs ....................................... 34
Figure 2-16 Algorithm of Function BasicElement ........................................... 36
Figure 2-17 Algorithm of Function MergeAggreValue .................................... 37
Figure 2-18 Algorithm of Function MergeAggrePointer .................................. 40
Figure 2-19 Algorithm of Function In .............................................................. 41
Figure 2-20 Algorithm of Function Equal................................................................. 43
Figure 2-21 Algorithm of Function GetElements....................................................... 45
Figure 2-22 Algorithms of Function MatchParenthesis and MatchQuote................. 46
Figure 2-23 Algorithm of Function CompleteRelation............................................. 48
Figure 2-24 the EXPRESS Schema for Target Database Manipulation....................... 50
Figure 2-25 Algorithms of Function dbDelete and dbShift ..................................... 53
Figure 2-26 Algorithm of Function dbUpdateReferences......................................... 55
Figure 2-27 Algorithm of Function dbCopyAttrs...................................................... 57
Figure 2-28 Algorithms of Function dbGetRootName and dbGetRootDef.................... 59
Figure 2-29 Algorithm of Function dbFindChild...................................................... 60
Figure 2-30 Algorithm of Function GetOIDs ............................................................ 60

Chapter 3 Generic Examples of the Internal Merge

Figure 3-1 Schema of Inheriting Example................................................................. 63
Figure 3-2 Part21 Data File of Inheriting Example..................................................... 63
Figure 3-3 the Target Data......................................................................................... 64
Figure 3-4 the Target Data......................................................................................... 65
Figure 3-5 the Final Target Data................................................................................. 66
Figure 3-6 the Final Target Data................................................................................. 66
Figure 3-7 Schema of an Example of Dependent Relations........................................ 67
Figure 3-8 An Example of Dependent Relations....................................................... 68
Figure 3-9 the Target Data......................................................................................... 70
Figure 3-10 the Target Data....................................................................................... 70
Chapter 4 Cost Advantage

Figure 4-1 CA Modeler and Cost Advantage .................................................. 79

Figure 4-2 an Example of a Report File .......................................................... 81

Figure 4-3 an Example of a Design File .......................................................... 82

Figure 4-4 CA Data Schema ......................................................................... 84

Figure 4-5 Algorithm of Function ReadFile .................................................. 85

Figure 4-6 Algorithm of Function ReadComponent ...................................... 87

Figure 4-7 Algorithm of Function ReadAssembly ......................................... 88

Figure 4-8 Algorithm of Function InsertComponent ...................................... 89

Figure 4-9 Algorithm of Function InsertAssembly ........................................ 90

Figure 4-10 Flow Chart of the Two Translators and Internal Merge ............... 91

Figure 4-11 Part21 Data Files of CA Example ............................................... 93

Figure 4-12 the Target Data File .................................................................... 94
Figure 4-13 the Target Data File ................................................................. 95

Figure 4-14 the Final Target Data File ......................................................... 96
Abbreviations

CIM – Computer Integrated Manufacturing

DBMS – DataBase Management System

RDBMS – Relational DataBase Management System

OODBMS – Object Oriented DataBase Management System

EQL – EXPRESS Query Language

TNR – Template and Rule language

IMDE – Integrated Manufacturing Design Environment

ISO – International Standards Organization

CA – Cost Advantage

UDMM – Unified Data Meta-Model

OID – Object ID
Chapter 1
Introduction

1.1. Introduction

Computer integrated manufacturing (CIM) technology is evolving very rapidly. Many companies that could not previously afford the technology are now finding that CIM is within their financial grasp. Developments in computer hardware, software and networks have occurred at a rapid pace over the past decade. The widespread use of computer technology in manufacturing has led to definite gains in productivity, improvements in quality and shortened product delivery cycles. Demands on manufacturing companies have led them to invest heavily in computer solutions.

Due to the unremitting efforts of thousands of software developers, more and more powerful and intelligent software packages have been widely used in current industrial world. Early users of these technologies observed great productivity increases in each of these applications domains, resulting in some increase in organizational efficiency and throughput.

Today's enterprises typically employ multiple software systems, which are independently developed, locally administered, and different in logical or physical designs. However, adding these tools in an ad hoc manner has created the proverbial islands of automation within the manufacturing enterprise.

Current research tends to focus on a limited set of manufacturing domains. Industrial development is usually geared toward point to point integration of specific tools to meet a
current particular need. Therefore, a fundamental challenge in enterprise information management is the sharing of information for enterprise users across organizational boundaries. This requires a global view and an integration architecture that spans business and technical activities of an enterprise.

1.2. Four Basic Problems of Integration

One of the main problems when implementing the CIM concept concerns information integration. The integration of heterogeneous manufacturing application databases requires the solution of four categories of problems:

a) Data Representation

b) Commonality

c) Synchronization

d) Communication

A data representation scheme needs to be developed for representation of manufacturing information in a neutral form if manufacturing data is to be shared by multiple applications that have different and optimized data structures. Once such a scheme exists, the common information of these applications should be represented in the neutral database whose data structures are defined in the scheme. Since the same information may relate to multiple domains, the synchronization between the various application domains must be constructed for presenting a consistent view of the common manufacturing information. And the communication between these heterogeneous
applications must be established to exchange information efficiently after a consistent view of the common information is developed.

The communication problem has been well addressed in literature. Research has also resulted in commercial standards that are implemented in all major operating systems for the communication of information over heterogeneous networks. Therefore, communication among the manufacturing databases is not an issue and will not be discussed in this thesis. Data representation, commonality and synchronization of manufacturing information have been and continue to be active areas of research. While very few research efforts address general integration solutions, the majority targets specific applications or restricted manufacturing domains to simplify the problem. These efforts are valuable in the restricted domains they address, but are limited in the contribution to the overall throughput of the enterprise. The following section highlights some of the noteworthy research efforts.

1.3. Literature Review

[1] and [2] in the bibliography section provide the general concepts of CIM systems and help the reader to design and implement such systems, such as the development and evaluation of the theories, models, problem-solving strategies, and computer programs.

The issue of data representation and commonality were addressed in [3] and [4]. [3] presents an architecture for integration using a common data representation scheme and an integration methodology based upon the synchronization of the various tools. This paper presents a high-level conceptual architecture for the integration of manufacturing
databases. [4] presents the development of a Unified Data Meta-Model (UDMM) for the representation of manufacturing information in a neutral format. An IDEF1X relational model was developed to represent information that is common among various application domains. The UDMM facilitated the neutral representation of manufacturing information such that translators from the various manufacturing domains could either access or update this information. These two papers were written by other members in our research team.

Current research and achievements in manufacturing data integration were addressed in [5], [6], [7], [8] and [9]. [5] gives an overview of the development of the field of Knowledge Engineering over the last 15 years. It describes three modeling frameworks that are supplemented by discussing some important methodological developments in more detail. A quality information system framework for the mechanical component manufacturing industry is presented in [6] at generic and specific levels. An object modeling technique methodology, which makes both the documentation and the transition from modeling to operational software easier, was discussed in [7]. [8] provides an abstract, generic and flexible methodology for integrating the product data management and a true open system architecture is provided at the end of this paper. Issues on generalization-based data mining in object-oriented databases are investigated in [9] with regard to three aspects: generalization of complex objects, class-based generalization, and extraction of different kinds of rules.

[10] presents an architecture for the synchronization of heterogeneous manufacturing databases and algorithms to update a local database with the consistent external view of
the data. In [11], the manufacturing systems design problem is characterized and an architecture is presented which supports manufacturing engineering in a complex, data-rich environment.

1.4. The IMDE Architecture

[11] presents an architecture, Integrated Manufacturing Design Environment (IMDE), which will maintain and verify constraints and relationships among the multiple subsystem models to address the needs and special requirements placed upon the design process for the integration of manufacturing systems. The IMDE environment has two components: a unified data meta-model and intelligent interfaces.

1.4.1. Unified Data Meta-Model

A key component to the functioning of the IMDE architecture is the unified data meta-model, abbreviated UDMM. It represents the types of data entities, relationships among these entities and common constraints of manufacturing systems design methods. The format for the data entities, the attributes and the relations it contains can be specified in the UDMM. Thus the UDMM can define a standardized data format for translating between different databases having different data formats. Although the UDMM can represent every manufacturing data entity, attribute and relation, it is not feasible, nor necessary, for the UDMM to specify all the data and relations for all methods and tools for manufacturing design. Only those entities, attributes, and relationships common to more than one manufacturing domain needs to be defined in the
UDMM. In addition to the common data in the UDMM, individual design tools will have data that is not specified in the UDMM, as necessary, to perform their particular functions. All the common information that overlaps between independent tools must be captured by the UDMM.

1.4.2. Intelligent Interface

The intelligent interface, which is another important component of the IMDE, functions as an intelligent agent for the local database. Each intelligent interface is required to understand its tool's local data model and their relationship to the UDMM. Additionally, the ability for sharing data from the local model with other tools is also required for each intelligent interface. To these requirements, the knowledge of the relationship between local tools and the UDMM, and the methods to translate them into a neutral format specified by the UDMM was added. All information exchanged between intelligent interfaces is in a neutral format specified by the UDMM.

The intelligent interface translates data from the local format to the neutral format and it must include translate data and access local data modules. Essentially, the translate data module is designed to solve the synchronization problem in the four integration issues addressed in section 1.2. The access local data module is specific for each tool, depending on how the local data is stored (e.g. relational database, object database, flat file, text file, etc.).
1.5. Thesis Scope

Synchronization of the heterogeneous databases may be classified into two different categories: 1) the assimilation of multiple application domain views into a consistent system view and 2) the update of an individual application domain with information from the consistent system view. For discussion purposes, the former is referred to as an external merge and the later is referred to as an internal merge.

This thesis focuses on the implementation methodology for internal merge, based on the general algorithms developed by Robert Judd and Dinesh Dhamija (addressed in [10]) and the IMDE architecture (addressed in section 1.4) of the intelligent interface.

The implementation details of a specific access-local-data module of the intelligent interface will be illustrated later in this thesis. This module consists of two translators that are used to exchange data between the consistent neutral database and the local database of a manufacturing cost-estimation software – Cost Advantage.

1.6. Thesis Organization

After a brief introduction of the background knowledge in chapter 1, this thesis provides a detailed explanation of the internal merge algorithms and practical testing examples in chapter 2, 3, and 4, followed by conclusions in chapter 5.

The main body (chapter 2, 3, and 4) of this thesis includes two major parts:

1) The implementation of internal merge is illustrated in chapter 2, which is also the focus area of this thesis. In chapter 2, the architecture of internal merge is illustrated first and the explanation of the design of merge rules follows. Base on these merge
rules, the overall internal merge algorithms are developed. Then the software implementation strategies are illustrated in detail. At last, the algorithms and the software implementation for operating the target database are presented.

2) After the illustration of all the internal merge algorithms, in order to make the abstract algorithms to be easily and intuitively accepted, several simple examples are presented in chapter 3. Chapter 4 provides a practical example of internal merge, which discusses how to get the cost information of different manufacturing processes automatically from a software package, named Cost Advantage, by means of two translators and how to merge this cost information.
Chapter 2

Internal Merge

2.1. Introduction

To synchronize an application domain with information from other tools in the manufacturing domain, external merge takes UDMM sub-set representations of manufacturing information from the other tools like CA, Unigraphics and Part. A consistency check is performed on these UDMM subsets. If inconsistencies are detected, an error is generated and the merge is terminated. If the representations are found consistent, an ISO10303/Part21 data file representing the consistent view is generated. This Part 21 file is then used by the internal merge algorithms to merge the consistent view of information in the manufacturing domain into the specific application domain requesting an information update.

The consistent view of information that is generated by the external merge may be different from the specific application domain data. If the data from the consistent view of information is simply copied to the local database of the specific application domain, some information that the user previously added to the local database will be lost after the update. The internal merge is used to merge these two data sets without losing the information from either one of them.
2.2. Architecture

Internal merge requires two object oriented data structures as input: 1) the source, which is the consistent view of the information in the manufacturing domain, and 2) the target, which is the requesting tool’s view of the information contained within its local database. Both the source and target are Part 21 files that are based upon EXPRESS schemas. The source schema contains those elements that are relevant to the source tools. The target schema represents all elements of the target tools’ database. The target schema has embedded within it, a subset of the UDMM containing those elements whose data are being merged with the source. As a result of performing an internal merge, the target is updated with a unified view based upon merge rules that are established. Figure 2-1 represents the internal merge architecture graphically.

![Figure 2-1 Internal Merge Architecture](image-url)
2.3. Merge Rules

The source contains information elements that are subsets of the UDMM schema. The target on the other hand contains an entire representation of the requesting tools' local database. As a result of the internal merge, the new target will contain all elements that were originally in the target, but not in the source without modification. The representation of the elements that appear in both, the source and the target, are determined based upon a pre-determined set of rules.

In order to accommodate a generalized solution which will allow the users to configure the merge based upon their specific application, a set of merge rules needs to be developed. These merge rules stipulate how the internal merge algorithms merge the various elements of the object-oriented database to produce the resultant target. Given that object oriented databases, particularly the Part 11 specification, represents entities, relations and attributes, the merge rules represent these elements of the UDMM subset of the target. The merge rules are themselves specified in a Part 11-based format. If entities or attributes in the schema of the target do not appear in the merge rules, then these entities or attributes will not be merged. This design allows for the representation of "output only" entities and attributes within the target schema. At the highest level, merge rules are identified by a rule name and a set of entities. The EXPRESS schema of merge rules is presented in Figure 2-2.
ENTITY EntityRule
ABSTRACT SUPERTYPE OF (ONEOF(AttribEntityRule, ChildEntityRule));
  Name : STRING;
  Relations : SET OF RelationRule;
  InfoAttribs : SET OF AttribRule;
  Children : SET OF ChildEntityRule;
UNIQUE
  key : name;
END_ENTITY;

ENTITY AttribEntityRule
SUPERTYPE OF (RootEntityRule)
SUBTYPE OF (EntityRule);
  inheritanceMergeRule: MergeRuleType;
  keyAttribs : SET OF AttribRule;
END_ENTITY;

ENTITY RootEntityRule
ABSTRACT SUPERTYPE OF (ONE OF (IndependentEntityRule, DependentEntityRule ))
SUBTYPE OF (AttribEntityRule);
  instanceMergeRule: MergeRuleType;
END_ENTITY;

TYPE MergeRuleType = ENUMERATION OF (source, union);
END_TYPE;

ENTITY DependentEntityRule
SUBTYPE OF (RootEntityRule);
  dependencies: SET[1:?] OF DependentRule;
END_ENTITY;

ENTITY IndependentEntityRule
SUBTYPE OF (RootEntityRule);
END_ENTITY;

ENTITY ChildEntityRule
SUBTYPE OF (EntityRule);
END_ENTITY;

ENTITY DependentRule;
  DefiningEntity : EntityRule;
  entityContainingPtr: EntityRule;
  relationDef : RelationRule;
END_ENTITY;

ENTITY RelationRule;
  AttribName : STRING;
  TerminalEntity : EntityRule;
  Cardinality : Cardinality;
  relationMergeRule: MergeRuleType;
END_ENTITY;

Type Cardinality = ENUMERATION OF (singleRef, setRef, bagRef, listRef, arrayRef);
END_TYPE;

ENTITY AttribRule;
  Name : STRING;
  Def : AttribDef;
END_ENTITY;

ENTITY AttribDef
ABSTRACT Supertype of {oneof(SimpleRule, AggregateRule, PointerRule)};
END_ENTITY;
The upper level elements in the schema are entities. Entities are composed of attributes which may be of type entity, aggregate (sets, bags, lists, arrays) or simple (integer, real, string, boolean, logical, binary, number) attributes. Some of these attributes are pointers to other entities.

There are two types of *EntityRule*: 1) *AttribEntityRule* and 2) *ChildEntityRule*. *AttribEntityRule* represents an entity that is an attribute of another *EntityRule* and *ChildEntityRule* represents an inheriting entity. *RootEntityRule* is the highest parent of
the inheriting entities. RootEntityRule can be furthermore divided into DependentEntityRule and IndependentEntityRule. DependentEntityRules represent entities, whose existence depends upon the existence of one or more instances of other entities. Entities whose instances are required for an instance of the dependent entity to exist will be called defining entities. IndependentEntityRules represent entities whose existences are independent of other entities. All the other entities in the merge rule schema define the attributes and relations that belong to the above EntityRules.

In EntityRule, name is the name of the EntityRule, children points to all the ChildEntityRules inheriting from it. EntityRules include two categories of relations: dependencies and relations. dependencies are those relationships that defining entities use to define their dependent entities. All other relations between entities are represented as a relation attribute in the EntityRule and are termed non-identifying relations. There are two types of other attributes in EntityRules: infoAttribs and keyAttribs. keyAttribs are those that are used to distinguish between two entity instances that belong to the same type of EntityRule. All other attributes are represented as infoAttribs.

So EntityRules have four categories of attributes: 1) Attributes that store pointers for dependencies (identifying relations), 2) attributes that store pointers for non-identifying relations, 3) attributes that are key attributes, and 4) other information attributes. Dependent entities may possibly have no keys defined. It is assumed that to uniquely identify an entity instance in the OODB structure, it is adequate to identify the instances of all the defining entities and the value of the key attributes of the current instance. The organization of the entity rules into the two categories of relations and four categories of
attributes enable the merge algorithms to utilize the structure to merge the source into the target efficiently.

The attribute `instanceMergeRule` and `inheritanceMergeRule` are enumeration of source or union. If source is specified for an entity, then the source entity instances are simply copied over into the target. If union is specified, then the target output will contain a union of the entity instances that appear in the source and the target. The `relationMergeRule` in `RelationRule` has similar meaning but it is for merging relations of entities in the source and target.

`AttribRule` is used to specify all the `keyAttribs` or `infoAttribs`. It can be classified into three categories: 1) `SimpleRule` (like integer or real or string...), 2) `PointerRule` (points to other `AttribEntityRule`), and 3) `AggregateRule` (aggregate of `SimpleRules` or `PointRules`).

As mentioned at the beginning of this section, the merge rules define all the common information of the source and target database structure, which is needed for internal merge, such as different type of entities, relations between entities, attributes of entities and inheritance of entities. The instantiated data file of the merge rules will contain all the information. The internal merge algorithm is based on this information.

2.4. Overall Approach

Given the merge rules described above, the process of conducting an internal merge is shown in Figure 2-3. Each rectangle in this figure represents a major function of the internal merge. The number in parentheses in each rectangle and the cylinder indicates the section where each function is explained.
It begins with a call to function `MergeDatabase` to merge the source with the target. `MergeDatabase` calls `MergeAllEntities` to merge all the entity instances from the source with the ones from the target. Then for each of these source instances, `MergeEntityInstance` first calls `Find` to get its corresponding target instance by comparing their key attribute values and dependent relations. Once corresponding entities are located, the algorithm calls `MergeInfoAttribs` to merge the information attributes of the independent entities, followed by the dependent entities whose defining entities have already been merged. All the `ChildEntityRules` and `AttribEntityRules` of an independent or dependent entity are merged simultaneously. If `Find` can not locate a corresponding entity instance in the target, `CreateEntity` is called to create a new entity instance in the target with all of its attributes identical to the source instance. During this process, the attributes describing the non-identifying relations are ignored. The merge process continues until all entities are merged. Finally, the non-identifying relations are merged for all entities in `CompleteRelation`. 
2.5. Internal Merge Algorithms

The internal merge algorithms, which merges the source and target databases, are based on the merge rules described above. The merge algorithms at each level of the hierarchy are presented in the following sub-sections.

2.5.1. Merge the Databases

Internal merge begins with a call to merge the databases, which in turn successively calls the merge algorithms for the other elements in the database.

2.5.1.1. Function MergeDatabase

MergeDatabase is the highest level function of internal merge. It involves merging all the entities listed in merge rules followed by merging all the non-defining relations in merge rules. Figure 2-4 presents its algorithm. It has no parameters.

```
MergeDatabase()
{
    MergeAllEntities()
    E = MergeRules.EntityRule
    FOR EACH e IN E DO
        FOR EACH r IN e.relations DO
            CompleteRelation(e.name, r.attribname, r.cardinality, r.relationMergeRule)
        ENDFOR
    ENDFOR
}
```

**Figure 2-4 Algorithm of Function MergeDatabase**

It first calls function MergeAllEntities to merge all the entity instances of the source with the target. Variable E stores all the EntityRules. Each EntityRule will be used to search data and get the information of its non-defining relations. Then function
CompleteRelation is called to merge these relations of every corresponding target and source instance that are not completed in MergeAllEntities.

2.5.1.2. Function MergeAllEntities

Function MergeAllEntities goes through all entities in the merge rules data file and merges all the listed entities. Figure 2-5 presents its algorithm. It has no parameters.

```plaintext
MergeAllEntities()
{
  E = [ALL IndependentEntityRule]
  M = []
  WHILE (E IS NOT EMPTY) DO
    FOR EACH e IN E DO
      QUERY THE SOURCE FOR S = {sourceOID: sourceOID IS OF TYPE e.name}
      unMergedTargetOIDs = {targetOID: targetOID IS OF TYPE e.name}
      FOR EACH sourceOID IN S DO
        MergeEntityInstance(sourceOID, e.name, e.OID, &unMergedTargetOIDs, 0)
      END FOR
      M = M UNION GetTree(e)
      IF (e.instanceMergeRule == .source.)
        FOR EACH t IN unMergedTargetOIDs DO
          dbDelete(targetSchema, target, t, e.name)
        END FOR
    END FOR
    E = {e: e IS A DependentEntityRule AND e.dependencies[].definingEntity IS IN M
         AND e IS NOT in M}
  END WHILE
}
```

Figure 2-5 Algorithm of Function MergeAllEntities

First, it gets all the IndependentEntityRules and stores them in E. Then for each element in E, this function gets all the source and target entity instances that belong to the same type of IndependentEntityRule, stores all the source entity instances in variable S and all the target entity instances in variable unmergedTargetOIDs. Next, this function calls MergeEntityInstance to merge every source instance. In MergeEntityInstance, the merged target entity instances are deleted from unmergedTargetOIDs. Variable M keeps
track of the entities that have been merged. After the FOR loop for MergeEntitylnstance, because all the child entities of one EntityRule were also merged by MergeEntitylnstance, function GetTree is called to get all its children and put them into M. Now it is time to check if the instanceMergeRule is source. If it is, every element left in unmergedTargetOIDs is moved out from the target by calling function dbDelete. After all the IndependentEntityRules are merged, the next DependentEntityRule, whose defining entities are all merged, is queried to be merged. This process continues until all the instances of all EntityRules are merged.

2.5.1.3. Function GetTree

Function GetTree (Figure 2-6) is used to get all the child entities of an EntityRule from the merge rules data file.

```
GetTree(parentOID)
{
  R = parentOID
  IF parentOID has children
    C = {OIDs of all the children of parentOID}
    FOR EACH c IN C DO
      R = R UNION GetTree(c)
    ENDFOR
  ENDIF
  RETURN R
}
```

Figure 2-6 Algorithm of Function GetTree

Parameter parentOID points to the EntityRule in the merge rules data file, whose all children are wanted.

First, this function collects all the children one level down the inheritance tree. Then for each of them, it calls itself recursively to collect all other children in lower levels.
2.5.2. Merge Each Entity Instance

*MergeEntityInstance* is called by *MergeAllEntities* to accomplish entity merges. *FillInheritanceTree* is called by *MergeEntityInstance* to get the inheritance information of an entity instance. In this sub-section, these two functions are explained and the concept of the object id (OID) map is illustrated.

2.5.2.1. Function *MergeEntityInstance*

For each entity instance in the source, function *MergeEntityInstance* is called to merge it with the corresponding instance in the target. If a corresponding instance does not exist, it will create a copy of the source instance. Its algorithm is presented in Figure 2-7.

2.5.2.1.1. Parameters

Parameter *sourceOID* is the source entity instance id that needs to be merged. Parameter *rootName* and *rootOID* are the name and instance id (in the merge rules data file) of the *RootEntityRule* of *sourceOID*. Parameter *unMergedTargetOIDs* stores the same type target instances that have not been merged, one of them needs to be merged with the *sourceOID*. If parameter *noFind* is 1, functions *Find* is called to find the corresponding target instance of *sourceOID*. If it is 0, *Find* is not called because the corresponding target instance is passed in as the parameter *unMergedTargetOIDs*. 
MergeEntityInstance(sourceOID, rootName, rootOID, unMergedTargetOIDS, noFind)
{  
  FillInheritanceTree(source, sourceOID, rootName, rootOID, &sourceOIDTree[], &sourceNameTree[])

  IF noFind == 1  
    targetOID = unMergedTargetOIDS  
  ELSE  
    targetOID = Find(sourceOID, rootOID, &unMergedTargetOIDS)  
  ENDIF

  IF targetOID IS NULL  
    targetOID = CreateEntity(sourceOID, sourceOIDTree[], sourceNameTree[])  
    MapInsert(sourceOID, targetOID)  
  ELSE  
    MapInsert(sourceOID, targetOID)  
    FillInheritanceTree(target, targetOID, rootName, rootOID, &targetOIDTree[], &targetNameTree[])
  ENDIF  
  
  i = 1  
  WHILE i < #sourceOIDTree[] AND i < #TargetOIDTree[]  
    AND sourceOIDTree[i] == targetOIDTree[i]  
    i = i + 1  
  ENDFLASH  

  numCommonGenerations = i - 1

  IF sourceOIDTree[0].inheritenceMergeRule == .union.  
    IF (numCommonGenerations < #TargetOIDTree[] AND  
        numCommonGenerations < #sourceOIDTree[])  
      Error  
      EXIT  
    ENDFLASH  
  ENDIF  
  
  newTargetOID = dbShift(TargetSchema, target, targetOID, sourceNameTree[#sourceNameTree-1])  
  MapDelete(sourceOID)  
  MapInsert(sourceOID, newTargetOID)  
  targetOID = newTargetOID  
  ENDIF  
  ELSE // InheritanceMergeRule == .source.  
  IF (numCommonGenerations != #TargetOIDTree[] OR  
      numCommonGenerations != #sourceOIDTree[])  
    newTargetOID = dbShift(TargetSchema, target, targetOID, sourceNameTree[#sourceNameTree-1])  
  ENDIF  
  MapDelete(sourceOID)  
  MapInsert(sourceOID, newTargetOID)  
  targetOID = newTargetOID  
  ENDIF  

  MergeInfoAttribs(sourceOID, targetOID, &SourceOIDTree[])  
}  

Figure 2-7 Algorithm of Function MergeEntityInstance
2.5.2.1.2. Description

Function `MergeEntityInstance` first calls `FillInheritanceTree` to get all the inheritance information of `sourceOID`. Then, it calls `Find` to determine if an entity corresponding to `sourceOID` exists in the target database. If not, a new entity instance is created by function `CreateEntity`. `CreateEntity` will also update all the dependencies and copy the key and information attributes from the source instance. If a corresponding target entity exists, `MergeEntityInstance` compares its inheritance level with the source instance to see if a shift of the inheritance level is required. After all this, all the information attribute values of `sourceOID` and `targetOID` are merged by calling function `MergeInfoAttribs`.

When comparing the inheritance level of `sourceOID` and `targetOID` to decide how to shift the `targetOID` inheritance level, if the value of the `inheritanceMergeRule` in the `RootEntityRule` in the inheritance tree is union, there are three cases that need to be dealt with:

1. If the `targetOID` is in the lower level of the inheritance tree, and `sourceOID` is in the higher level, in this case, `targetOID` contains more attributes than `sourceOID`. The common attributes of `sourceOID` and `targetOID` are merged and the extra attributes that belong to `targetOID` only are untouched.

2. If the `targetOID` is in the higher level of the inheritance tree, while `sourceOID` is in the lower level, in this case, `sourceOID` contains more attributes than `targetOID`. Function `dbShift` is called to shift the `targetOID` to the same level as `sourceOID`. Then the new `targetOID` has the same attributes as the original `targetOID` and all the relations that reference the
original targetOID are changed to the new targetOID. Then the new targetOID is merged with the sourceOID.

(3) If the sourceOID and targetOID are in different branches of the inheritance tree, an error message is generated and the TNR code is terminated.

If the value of the inheritanceMergeRule in the RootEntityRule is source, targetOID is just shifted to the same inheritance level as sourceOID, then they are merged.

2.5.2.2. The OID Map

The unique identifier of the object instance in an OODBMS is the object id (OID). Essentially, the OID is a pointer that points to the data entity instance. By using this pointer, the entity data can be obtained instantly without joining several tables together by their foreign keys in a typical RDBMS way. This can save a lot of computer time in searching and make the implementation code much more efficient.

The OID in the source and target databases will be different for corresponding entities since they are generated by the OODBMS. Section 2.5.3 provides an algorithm and implementation for finding corresponding entities. While the algorithm is relatively simple, it can be tedious to navigate through the OIDs in the source and target databases. Thus, an OID map, which contains a map of the OIDs of the source and target databases will be instantiated and maintained. The procedures defined for the OID map are given in Figure 2-8. These three procedures can be implemented easily by means of EQL and TNR and their implementations are not discussed in this thesis.
MapInsert(sourceOID, targetOID) /* insert corresponding sourceOID and targetOID into OID map. */
MapSource(targetOID) /* Returns OID of corresponding instance in the source. */
MapTarget(sourceOID) /* Returns OID of corresponding instance in the target. */
MapDelete(sourceOID) /* delete sourceOID and its corresponding targetOID from the OID map. */

Figure 2-8 Procedures for the OID Map

An example of the implementation of OID map is in Figure 2-9. Every number following a pound sign is an OID. The left column represents some OIDs in the source database, and the right column represents some in the target. The map stores all the corresponding pairs of source and target OIDs.

| #891341473 | #733741388 |
| #107417419 | #578781093 |
| #715771834 | #741570997 |
| ...... | ...... |

Figure 2-9 an Example of OID Map

In function MergeEntityInstance, every time the corresponding targetOID is found or a new target instance is created or the inheritance level of a target instance is shifted, corresponding changes in the OID map need to be made.

2.5.2.3. Function FillInheritanceTree

Function FillInheritanceTree is used to find out all the names and EntityRule instance ids of the EntityRules that are higher or at the same inheritance level. It is called by function MergeEntityInstance to get the inheritance information of sourceOID and
targetOID to compare their inheritance levels. Figure 2-10 is the algorithm of this function.

```
void FillInheritanceTree(dbase, OID, rootName, rootOID, &OIDTree[], &nameTree[])
{
    OIDTree[0] = rootOID
    nameTree[0] = rootName;
    i = 0
    WHILE OIDTree[i].children IS NOT EMPTY DO
        C = [OIDTree[i].children]
        FOR EACH c IN C DO
            IF OID IS OF Type c.name IN dbase
                BREAK
            ENDIF
        ENDFOR
        IF c EXIST
            i = i + 1
            OIDTree[i] = c
            nameTree[i] = c.name
        ELSE
            BREAK
        ENDIF
    ENDFOR
}
```

Figure 2-10 Algorithm of Function FillInheritanceTree

2.5.2.3.1. Parameters

Parameter `dbase` is the name of the data file of the source or target. `OID` is the entity instance id in `dbase`. The names and instance ids (in the merge rules data file) of all the `EntityRules` at the same inheritance level or higher than the `EntityRule` corresponding to `OID` are returned by `OIDTree[]` and `nameTree[]` respectively. Parameter `rootName` and `rootOID` are the name and instance id (in the merge rules data file) of the `RootEntityRule` of `OID`. 
2.5.2.3.2. Description

Starting from the highest level of the inheritance tree, this function first puts the rootName and rootOID into array nameTree[] and OIDTree[]. Then it loops through every child of the RootEntityRule to see if OID is of this type. If it is, this function appends its name and instance id to nameTree[] and OIDTree[]. The function repeats this process it reaches the exact level of OID.

2.5.3. Find Corresponding Target Instance

The first step of the process of conducting an internal merge involves finding corresponding instances in the target database. Function Find and CheckDependencies are used to accomplish this task.

2.5.3.1. Function Find

This function is used to find the OID in target corresponding to a source instance. Figure 2-11 shows its algorithm.

2.5.3.1.1. Parameters

Parameter sourceOID is the OID in the source whose corresponding target OID is wanted. rootOID is the instance id (in the merge rules data file) of RootEntityRule of sourceOID. unmergedTargetOIDs is the same as in function MergeEntityInstance.
Find(sourceOID, rootOID, unMergedTargetOIDs)
{
  IF unMergedTargetOIDs == "" THEN
    RETURN NULL
  ENDIF
  A = {the key attributes of sourceOID}
  T = {}
  IF A IS NOT EMPTY
    FOR EACH u IN unMergedTargetOIDs DO
      e = 1
      FOR EACH a in A DO
        vs = value of a.name from sourceOID
        vt = value of a.name from u
        IF NOT Equal(vs, vt, a.def)
          e = 0
          BREAK
        ENDIF
      ENDFOR
      IF e==1
        T = T UNION u
      ENDIF
    ENDFOR
  ELSE
    T = {OID: OID IS IN unMergedTargetOIDs}
  ENDIF
  FOR EACH t in T
    IF CheckDependencies(sourceOID, t, rootOID) = 1
      REMOVE t FROM UnmergedTargetOIDs
      RETURN t
    ENDIF
  ENDFOR
  RETURN NULL
}

Figure 2-11 Algorithm of Function Find

2.5.3.1.2. Description

This function first finds all instances in unmergedTargetOIDs that have the same key attribute values of sourceOID (stored in A). All the instances found are stored in T. Then for each element in T, the function checks if the element has the same defining relations as sourceOID by calling function CheckDependencies. It returns the target OID if it finds one, otherwise, it returns “NULL”. Passing the value of unmergedTargetOIDs to this function narrows the search domain for the target instance.
2.5.3.2. Function CheckDependencies

Function CheckDependencies is called by Find to check if a source instance and a target instance have the same defining entities in every defining relation. Figure 2-12 shows the algorithm used.

```plaintext
CheckDependencies(sourceOID, targetOID, rootOID) {
    IF rootOID IS A DependentEntityRule
    D = {rootOID.depedencies}
    FOR EACH d IN D DO
        IF (d.definingEntity == d.entityContainingPtr) 
            IF (d.relationdef.cardinality == .singleRef.) THEN 
                QUERY source FOR AN INSTANCE OID1 OF TYPE 
                d.definingEntity.name WITH ATTRIBUTE 
                d.relationdef.attribName == sourceOID 
                targetInstance = mapTarget(OID1) 
                QUERY target FOR AN INSTANCE OID2 OF TYPE 
                d.definingEntity.name WITH ATTRIBUTE 
                d.relationdef.attribName == targetOID 
                IF(targetInstance != OID2) 
                    RETURN 0 
            ENDIF 
            ELSE 
                QUERY source FOR AN INSTANCE OID1 OF TYPE 
                d.definingEntity.name WITH ATTRIBUTE 
                d.relationdef.attribName CONTAINING sourceOID 
                targetInstance = mapTarget(OID1) 
                QUERY target FOR AN INSTANCE OID2 OF TYPE 
                d.definingEntity.name WITH ATTRIBUTE 
                d.relationdef.attribName CONTAINING targetOID 
                IF(targetInstance != OID2) 
                    RETURN 0 
            ENDIF 
    ENDIF 
    ELSE // d.definingEntity != d.entityContainingPtr 
        QUERY source FOR VALUE1 = d.relationdef.attribName 
            FROM sourceOID 
        targetInstance = mapTarget(VALUE1) 
        QUERY Target FOR VALUE2 = d.relationdef.attribName 
            FROM targetOID 
        IF(targetInstance != VALUE2) 
            RETURN 0 
    ENDIF 
ENDIF 
ENDFOR 
ENDIF 
RETURN 1 
}
```

Figure 2-12 Algorithm of Function CheckDependencies
2.5.3.2.1. Parameters

Parameter sourceOID is the source instance id. targetOID is the target instance id. Parameter rootOID is the same as in function Find.

2.5.3.2.2. Description

This function returns 1 if sourceOID and targetOID have the same dependencies, 0 otherwise. All the dependent relations of RootEntityRule are queried first from rootOID. Then for each dependent relation, this function checks if it is identical for sourceOID and targetOID.

To compare a dependent relation, it is required to determine whether the pointer of the relation is contained in the current instance or in its defining entity instances. If the defining entities contain the pointer, their instance ids are queried from the source and target database. These instances are termed OID1 and OID2 respectively. Then the map is searched to find the corresponding target OID for OID1. This is compared with OID2 to see if they are equal. Since the order of internal merge is always the defining entity first, so by the time a dependent relation of some entity instance is compared, its defining entity OID must already be put into the map. This means OID1 and its corresponding target OID must be in the map.

If sourceOID and targetOID contain the pointer of the defining relation, this function just gets the value of this pointer in sourceOID and targetOID and compares them. By passing the three OIDs to function CheckDependencies, a better searching efficiency can be attained in this function's queries.
After all the defining relations are compared, this function returns 1 if they are equal. If any one of them is not equal, this function returns 0 instantly.

2.5.4. Create Entity Instance

When a source entity instance does not have a corresponding target instance, function CreateEntity and CreateAggreEntity are called to create it in the target. These two functions are discussed in this section.

2.5.4.1. Function CreateEntity

Function CreateEntity is called by MergeEntityInstance in case Find fails. It is used to create a new entity instance in the target database with all its attribute values identical to the sourceOID of MergeEntityInstance. Figure 2-13 presents the algorithm used.

2.5.4.1.1. Parameters

sourceOID is the OID in the source, whose identical entity instance needs to be created in target. OIDTree[] and nameTree[] are the arrays that contain the information of sourceOID inheritance tree.

2.5.4.1.2. Description

This function first goes through every element in OIDTree[] to collect all the names of the key attributes and information attributes, stores them into array A[]. Next, it selects all the values of these attributes from sourceOID and stores them into array V[].
The FOR loop is used to filter out all the names and values of PointerRules or aggregate of PointerRules from $A[]$ and $V[]$. A new entity instance is created with the same attribute values left in $V[]$.

```c
CreateEntity(sourceOID, OIDTree[], nameTree[])
{
    A[] = {OIDTree[0].keyAttribs}
    FOR (i=0; i<#OIDTree[]; i++)
        A[] = A[] UNION OIDTree[i].infoAttribs[]
    ENDFOR

    V[] = {values OF attributes IN A FROM sourceOID}
    FOR i=0 TO #A[] DO
        IF A[i] IS A pointer attribute THEN
            APPEND V[i] TO ARRAY pointerValue[]
            APPEND A[i] TO ARRAY pointerAttribRule[]
            DELETE V[i] FROM V[]
            DELETE A[i] FROM A[]
        ENDIF
    ENDFOR

    targetOID = INSERT INTO target an Entity of Type nameTree[maxNumber]
    WITH ATTRIBUTE VALUES V[]

    IF OIDTree[0] IS A type of DependentEntityRule THEN
        D = {OIDTree[0].dependencies}
        FOR EACH d IN D DO
            IF (d.definingEntity == d.entityContainingPtr) THEN
                IF (d.relationdef.cardinality == .singleRef.) THEN
                    QUERY source FOR an instance OID of type d.definingEntity.name WITH attribute d.relationdef.attribName == sourceOID
                    targetInstance = mapTarget(OID)
                    UPDATE targetOID, SET d.relationdef.attribName = targetOID
                ELSE
                    QUERY source FOR the VALUE of d.relationdef.attribName FROM sourceOID
                    UPDATE targetOID, SET d.relationdef.attribName = mapTarget(VALUE)
                ENDIF
            ELSE
                QUERY source FOR an instance OID of type d.definingEntity.name WITH attribute d.relationdef.attribName CONTAINING sourceOID
                targetInstance = mapTarget(OID)
                UPDATE targetInstance, ADD targetOID TO d.relationdef.attribName
            ENDIF
        ENDFOR

        FOR i=0 to #pointerAttribRule[]-1 DO
            pointerValue[i] = CreateAggreEntity(pointerValue[i], pointerAttribRule[i].def)
        ENDFOR

        UPDATE targetOID SET pointerAttribRule[].name = pointerValue[]
    RETURN (targetOID)
}
```

Figure 2-13 Algorithm of Function CreateEntity
Now it is necessary to make the dependent relations of \textit{targetOID} and \textit{sourceOID} identical to each other. For each dependent relation, if the defining entity contains the pointer, this function queries the source to find the defining entity instance, searches the map to find its corresponding target instance and updates this instance to make it point to \textit{targetOID}. The pointer of dependent relations can be an aggregate value. In that case, this function just adds the \textit{targetOID} to the defining entity instance in the target database. If \textit{sourceOID} and \textit{targetOID} contain the pointer of a dependent relation, this function gets the value of that relation from \textit{sourceOID} and sets the \textit{targetOID} with the value corresponding to the value of \textit{sourceOID} in the map.

\subsection*{2.5.4.2. Function \textit{CreateAggreEntity}}

At the end of function \textit{CreateEntity}, function \textit{CreateAggreEntity} is called to create all the instances of \textit{AttribEntityRules} in all the \textit{PointerRule} or aggregate of \textit{PointerRules} attributes of the \textit{sourceOID}. Figure 2-14 presents the algorithm of function \textit{CreateAggreEntity}.

\begin{verbatim}
CreateAggreEntity(attribValue, attribDef)
{
    IF attribDef IS A PointerRule THEN
        FillInheritanceTree(source, attribValue, attribDef.pointer.name,
               attribDef.pointer, &OIDTree[], &nameTree[])
        RETURN CreateEntity(attribValue, &OIDTree[], &nameTree[])
    ELSE // attribRule IS A AggregateRule
        GetElements(attribValue, elements[]) 
        R = ""
        FOR i=0 TO #elements[]-1 DO
            R = "," | CreateAggreEntity(elements[i], attribDef.elementDef)
        ENDFOR
        DELETE the first comma from R and put it into {}.
        RETURN R
    ENDIF
}
\end{verbatim}

\textbf{Figure 2-14 Algorithm of Function \textit{CreateAggreEntity}}
2.5.4.2.1. Parameters

\textit{attribValue} is the value of the attribute whose basic element is \textit{PointerRule}. \textit{attribDef} is the definition of this attribute (same as \textit{attribDef} in the merge rules schema).

2.5.4.2.2. Description

This function first checks if \textit{attribValue} is a single \textit{PointerRule} or aggregate of \textit{PointerRules}. If it is a single \textit{PointerRule}, this function simply calls \textit{CreateEntity} to create a instance of \textit{AttribEntityRule}. If it is an aggregate of \textit{PointerRules}, this function calls \textit{GetElements} to break \textit{attribValue} one level down and puts all the elements into \textit{elements[]}. Then, for each element, this function calls itself recursively to create all the instances of \textit{AttribEntityRule} in that element. In the end, it returns the corresponding value of \textit{attribValue}.

2.5.5. Merge Information Attributes

At the end of function \textit{MergeEntityInstance}, function \textit{MergeInfoAttrs} is called to merge the information attributes of the matched source and target instances. It calls several other functions to accomplish the merge. All of these functions are discussed hierarchically in this section.

2.5.5.1. Start to Merge \textit{infoAttrs}

The merge of \textit{infoAttrs} begins with a call to function \textit{MergeInfoAttrs}, which calls function \textit{BasicElement}. These two functions are discussed in this section.
2.5.5.1.1. Function \textit{MergeInfoAttribs}

The algorithm of \textit{MergeInfoAttribs} is shown in Figure 2-15. This function merges all the information attributes of \textit{sourceOID} and \textit{targetOID} and puts the merged value into \textit{targetOID}.

\begin{verbatim}
MergeInfoAttribs(sourceOID, targetOID, OIDTree[])
{
    A={};
    FOR (i=0; i<#OIDTree[-1]; i++)
        A = A UNION {OIDTree[i].infoAttribs[]}
    ENDFOR

    FOR EACH a IN A DO
        vs = values of a.name IN sourceOID
        IF a.def IS A PointerRule THEN
            vt = values of a.name IN targetOID
            IF Equal(vs, vt, a.def)
                MergeEntityInstance(vs, a.def.pointer.name, a.def.pointer, vt, 1)
                CONTINUE
            ELSE
                dbDelete(TargetSchema, target, vt, a.def.pointer.name)
                FillInheritanceTree(source, vs, a.def.pointer.name, a.def.pointer,
                    &OIDTree[], &nameTree[])
                vs = CreateEntity(vs, OIDTree[], nameTree[])
            ENDIF
        ENDIF

        IF a.def IS A RootAggregateRule THEN
            vt = values of a.name IN targetOID
            IF BasicElement(a.def) == 1 // basic element are pointers
                IF a.def.AggregateMergeRule == .union. THEN
                    vs = MergeAggrePointer(vs, vt, a.def)
                ELSE // a.def.AggregateMergeRule == .source.
                    elements[] = all the values of pointers IN vt
                    FOR i=0 to #elements[]-1 DO
                        dbDelete(TargetSchema, target, elements[i], 'NULL')
                    ENDFOR
                    vs = CreateAggreEntity(vs, a.def)
                ENDIF
            ELSE // basic element are simple values.
                IF a.def.AggregateMergeRule == .union. THEN
                    vs = MergeAggreValue(vs, vt, a.def)
                ENDIF
            ENDIF
        ELSE
            UPDATE TargetOID (SET a.name = vs)
        ENDFOR
}
\end{verbatim}

Figure 2-15 Algorithm of Function \textit{MergeInfoAttribs}
2.5.5.1.1. Parameters

Parameter $OIDTree[]$ is the array storing all the inheritance tree information of $sourceOID$ and $targetOID$. Note that by the time this function is called, $sourceOID$ and $targetOID$ must be at the same inheritance level.

2.5.5.1.1.2. Description

Function $MergeInfoAttribs$ first goes through all the inheritance levels in $OIDTree[]$ to collect all the information attributes and stores them in variable $A$. Then, for each element in $A$, it selects the value of this attribute from $sourceOID$ and stores it in variable $vs$ (Value of Source).

If this attribute is a $PointerRule$ attribute, which means the value of this attribute is a pointer that points to an $AttribEntityRule$ instance, function $MergeInfoAttribs$ selects the value of this attribute from $targetOID$, stores it in variable $vt$ (Value of Target) and calls $Equal$ to see if $vt$ and $vs$ are identical (if the two $AttribEntityRule$ instances that $vs$ and $vt$ point to have the same key attribute values). If they are, this function calls $MergeEntityInstance$ to merge the two $AttribEntityRule$ instances that $vs$ and $vt$ point to. Otherwise, this function calls $dbDelete$ to delete the $AttribEntityRule$ instance that $vt$ points to and calls $CreateEntity$ to create an entity instance in $target$ which is identical to the one that $vs$ points to.

If the attribute is an aggregate attribute, this function calls $BasicElement$ to see if the basic element of this aggregate attribute is $PointerRule$. If it is and $AggregateMergeRule$ is union, this function calls $MergeAggrePointer$ to merge all the $PointerRule$ instances in
vs with vt. If the basic element is PointerRule and AggregateMergeRule is source, this function calls dbDelete to delete all the AttribEntityRule instances of vt and calls CreateAggreEntity to create all the AttribEntityRule instances of vs in the target. If the basic element is not PointerRule and AggregateMergeRule is union, this function calls MergeAggreValue to merge vs with vt.

For all other cases, this function just updates the attribute of targetOID with vs.

2.5.5.1.2. Function BasicElement

Function BasicElement is called by MergeInfoAttribs to see if the basic element of an aggregate value is PointerRule. If it is, function BasicElement returns 1, otherwise returns 0. Its algorithm is presented in Figure 2-16.


def BasicElement(attribDef):
    while attribDef.elementDef IS A AggregateRule:
        attribDef = attribDef.elementDef
    if attribDef.elementDef IS A SimpleRule:
        return 0
    else:
        return 1

Figure 2-16 Algorithm of Function BasicElement

Parameter attribDef is the definition of an aggregate attribute (same as attribDef in the merge rules schema).

This function traces the attribDef of the attribute until it reaches the lowest level element and checks to see if it is a SimpleRule or not.
2.5.5.2. Merge Attribute Values

Two functions, \textit{MergeAggreValue} and \textit{MergeAggrePointer}, are used to merge the values of an aggregate attribute. They are discussed in this section.

2.5.5.2.1. Function \textit{MergeAggreValue}

Function \textit{MergeAggreValue} is used to merge two attribute values from the source and target and the attribute is aggregate/aggregates of \textit{SimpleRules}. The algorithm of function \textit{MergeAggreValue} is presented in Figure 2-17.

\begin{verbatim}
MergeAggreValue(s, t, attribDef)
{
    m = ""
    SWITCH (attribDef.type)
    CASE array:
        m = s
    CASE set:
        CASE bag:
            GetElements(s, source[]);
            GetElements(t, target[]);
            FOR(i=0; i<#target[]; i++)
                IF (p = In(target[i], source[], attribDef, 1)) >= 0 THEN
                    REMOVE source[p] FROM source[]
                ENDIF
            ENDFOR
            m = target[] union source[]
        ENDcase
        CASE list:
            GetElements(s, source[]);
            GetElements(t, target[]);
            FOR(i=0; i<#target[]; i++)
                IF (p = In(target[i], source[], attribDef, 1)) >= 0 THEN
                    FOR j = 0 TO p
                        m = m UNION source[j]
                    REMOYE source[j] FROM source[]
                    ENDFOR
                ELSE
                    m = m UNION target[i]
                ENDIF
            ENDFOR
            m = m UNION source[]
    ENDswitch

    RETURN m
}
\end{verbatim}

Figure 2-17 Algorithm of Function \textit{MergeAggreValue}
2.5.5.2.1.1. Parameters

Parameter \( s \) and \( t \) are the values of the attribute from source and target respectively. \( \text{attribDef} \) is the value of \( \text{attribDef} \) of the attribute in the merge rules data file.

2.5.5.2.1.2. Description

There are four kinds of aggregate attribute: array, set, bag and list. For array, this function just returns the value of \( s \) as the merged value.

For set and bag, the merged value of \( s \) and \( t \) are the same type of aggregate containing all the elements in \( s \) and \( t \). In this case, function \( \text{GetElements} \) is called to put all the elements of \( s \) and \( t \) into array \( \text{source}[] \) and \( \text{target}[] \). Then for each element in \( \text{target}[] \), function \( \text{In} \) is called to see if it is an element in \( \text{source}[] \). If so, function \( \text{MergeAggreValue} \) removes this repeated element from \( \text{source}[] \). After the above FOR loop, \( \text{source}[] \) only contains the non-repeated elements. So the union of \( \text{target}[] \) and \( \text{source}[] \) is the merged value.

For list type, first every element of \( s \) and \( t \) is put into \( \text{source}[] \) and \( \text{target}[] \) respectively. Then for each element in \( \text{target}[] \), function \( \text{In} \) is called to see if it is an element in \( \text{source}[] \). If so, function \( \text{In} \) returns the index \( p \) of this element in \( \text{source}[] \), the elements from 0 to \( p \) are moved out from \( \text{source}[] \) and put into \( m \) which stores the final merged value of \( s \) and \( t \). Otherwise, element \( \text{target}[i] \) is put into \( m \) directly. In the end, this function appends the elements left in \( \text{source}[] \) to \( m \). This creates a merged list which retains the partial ordering implied by the target and source lists.
2.5.5.2.2. Function *MergeAggrePointer*

Function *MergeAggrePointer* is used to merge two attribute values from the source and target when the attribute is an aggregate of *PointerRules*. Its algorithm, which is similar to *MergeAggreValue*, is in Figure 2-18 and they have the same parameters.

For array, all the instances that are pointed by \( t \) are deleted from the target. Then all the instances that are pointed by \( s \) are copied from the source by calling function *CreateAggreEntity*.

```plaintext
MergeAggrePointer(s, t, attribDef)
{
    mergedValue = ""

    SWITCH (attribDef.type)
    CASE array:
        DELETE all the instances in t
        MergedValue = CreateAggreEntity(s, attribDef)
    CASE set:
    CASE bag:
        GetElements(s, source[]);
        GetElements(t, target[]);
        IF attribDef.elementDef IS A PointerRule THEN
            FOR(i=0; i<#target[]; i++)
                IF (p = In(target[i], source[], attribDef, 1)) >= 0 THEN
                    MergeEntityInstance(source[p], attribDef.elementDef.pointer.name, attribDef.elementDef.pointer, target[i], 1)
                    REMOVE source[p] FROM source[]
                ENDIF
                mergedValue = target[i]
            ENDFOR
            FOR(i=0; i<#source[]; i++)
                FillInheritanceTree(source[i], attribDef.elementDef.pointer.name, attribDef.elementDef.pointer, &OIDTree[], &nameTree[])
                mergedValue = CreateEntity(source[i], OIDTree[], nameTree[])"
            ENDFOR
        ELSE // merge set/bag of aggregate of pointers.
            FOR(i=0; i<#target[]; i++)
                IF (p = In(target[i], source[], attribDef, 1)) >= 0 THEN
                    MergeAggrePointer(source[p], target[i], attribDef)
                    REMOVE source[p] from source[]
                ENDIF
                mergedValue = target[i]"
            ENDFOR
            FOR(i=0; i<#source[]; i++)
                mergedValue = CreateAggreEntity(source[i], attribDef.elementDef)"
            ENDFOR
        ENDIF
    ELSE // merge set/bag of aggregate of pointers.
        FOR(i=0; i<#target[]; i++)
            IF (p = In(target[i], source[], attribDef, 1)) >= 0 THEN
                MergeAggrePointer(source[p], target[i], attribDef)
                REMOVE source[p] from source[]
            ENDIF
            mergedValue = target[i]"
        ENDFOR
        FOR(i=0; i<#source[]; i++)
            mergedValue = CreateAggreEntity(source[i], attribDef.elementDef)"
        ENDFOR
    ENDIF
    DELETE the last character of mergedValue and put it into the ()
```
CASE list:
GetElements(s, source[]);
GetElements(t, target[]);
IF attribDef.elementDef IS A PointerRule THEN
FOR(i=0; i<$target[]; i++)
  IF { p = In(target[i], source[], attribDef, 1)) >= 0 THEN
    MergeEntityInstance(source[p], attribDef.elementDef.pointer.name,
    attribDef.elementDef.pointer, target[i], 1)
    FOR j = 0 to p-1
      FillInheritanceTree(Source, source[j], attribDef.elementDef.pointer.name,
      attribDef.elementDef.pointer, &OIDTree[], &nameTree[])
      mergedValue += CreateEntity(source[j], &OIDTree[], &nameTree[])
    ENDFOR
    REMOVE source[0 to p] FROM source[]
ENDIF
mergedValue += target[i]"
ENDIF
FOR(i=0; i<$source[]; i++)
  FillInheritanceTree(source, source[i], attribDef.elementDef.pointer.name,
  attribDef.elementDef.pointer, &OIDTree[], &nameTree[])
  mergedValue += CreateEntity(source[i], &OIDTree[], &nameTree[])
ENDFOR
ELSE //merge list of aggregate of pointers
FOR(i=0; i<$target[]; i++)
  IF { p = In(target[i], source[], attribDef, 1)) >= 0 THEN
    MergeAggrePointer(source[p], target[i], attribDef.elementDef)
    FOR j = 0 to p-1 DO
      mergedValue += CreateAggreEntity(source[j], attribDef.elementDef)
    ENDFOR
    remove source[0 to p] from source[]
ENDIF
mergedValue += target[i]"
ENDIF
FOR(i=0; i<$source[]; i++)
  mergedValue += CreateAggreEntity(source[i], attribDef.elementDef)
ENDFOR
ENDIF
DELETE the last character of mergedValue and put it into the ()
ENDSWITCH
RETURN mergedValue
} /* END MergeAggrePointer */

Figure 2-18 Algorithm of Function MergeAggrePointer

For set or bag, this function calls GetElements to put all the elements of s and t into source[] and target[] respectively. Then for each element of target[], this function calls In to see if a corresponding element exists in source[]. If it does, this function merges these two corresponding elements and move the one out of source[]. After this FOR loop, this function creates all the elements left in source[] into target and appends them to the merged value.
For list, this function does the same thing as in \textit{MergeAggreValue} except that each time this function calls \textit{In} and finds one corresponding element in \textit{source[]}, it merges the corresponding elements before it moves the element out of \textit{source[)}. And when this function needs to append the extra elements left in \textit{source[]}, it creates them in the target and appends the created target OIDs to the merged value.

\subsection*{2.5.5.3. Compare Values}

Function \textit{In} and \textit{Equal} are called by \textit{MergeAggreValue} and \textit{MergeAggrePointer} to compare attribute values. These two functions are discussed in this section.

\subsubsection*{2.5.5.3.1. Function \textit{In}}

Figure 2-19 presents the algorithm of function \textit{In}.

\begin{verbatim}
In(s, t[], attribDef, inverse)
{
    IF inverse == 1 THEN
        FOR(i=0; i<$t[]$; i++)
            IF Equal(t[i], s, attribDef.elementDef) THEN
            RETURN i
            ENDIF
        ENDFOR
    ELSE
        FOR(i=0; i<$t[]$; i++)
            IF Equal(s, t[i], attribDef.elementDef) THEN
            RETURN i
            ENDIF
        ENDFOR
    ELSE
        RETURN -1;
    }
\end{verbatim}

\textbf{Figure 2-19 Algorithm of Function In}

It is used to see if an element \textit{s} is in an array \textit{t[]} and return its index in \textit{t[]}.. \textit{attribDef} is the value of \textit{attribDef} of \textit{s} or the elements of \textit{t[]} in the merge rules data file. If \textit{inverse}
is 1, \( s \) is from the source and \( t[j] \) is from the target. If \( \text{inverse} \) is 0, \( s \) is from the target and \( t[j] \) is from the source. As the first parameter of \( \text{Equal} \) is always the value from the source and the second is always from the target, \( \text{inverse} \) is needed to decide the order of \( s \) and \( t[i] \) in function \( \text{Equal} \). If there is an element in \( t[j] \) which is equal to \( s \), this function returns its index, otherwise, returns –1.

2.5.5.3.2. Function \( \text{Equal} \)

Function \( \text{Equal} \) is called to judge if two values are equal. These two values can be simple values or aggregate values. Figure 2-20 presents its algorithm.

2.5.5.3.2.1. Parameters

Parameter \( s \) and \( t \) are any two values of the same attribute from the source and target database respectively. Parameter \( \text{attribDef} \) is the value of \( \text{attribDef} \) of the attribute in the merge rules data file. This function returns 1 if \( s \) equals \( t \), 0 otherwise.

2.4.5.3.2.2. Description

If \( s \) and \( t \) are simple values, this function compares them directly and returns the result. If \( s \) and \( t \) are values of \( \text{PointerRule} \), this function compares all the key attribute values (calls \( \text{Equal} \) recursively) of the \( \text{AttribEntitytRule} \) instances pointed to by \( s \) and \( t \). If \( s \) and \( t \) are arrays or lists, each element in \( s \) and \( t \) needs to be compared by calling \( \text{Equal} \) recursively. Only if \( s \) and \( t \) have the same elements, this function returns 1. If \( s \) and \( t \) are sets, this function first checks if they have the same number of elements, then if each
Figure 2-20 Algorithm of Function `Equal`

```plaintext
equal(s, t, attribDef) {
    SWITCH {attribDef.type}
    CASE SimpleRule(integer, real, logic, string):
        RETURN s==t
    CASE pointer:
        FOR i=0 TO #attribDef.pointer.keyAttribs[
            vs = value of keyAttribs[i].name from s
            vt = value of keyAttribs[i].name from t
            IF NOT Equal(vs, vt, keyAttribs[i].def)
                RETURN 0
            ENDIF
        ENDFOR
        RETURN 1
    CASE array:
        CASE list:
            GetElements(s, source[]); GetElements(t, target[]);
            IF (#source[] == #target[]) THEN
                FOR i=0 TO #source[]-1
                    IF NOT Equal(source[i], target[i], attribDef.elementDef)
                        RETURN 0 //arrays are not equal
                    ENDIF
                ENDFOR
                RETURN 1 //arrays equal
            ENDIF
        ENDIF
    CASE set:
        GetElements(s, source[]); GetElements(t, target[]);
        IF (#source[] == #target[]) THEN
            FOR i = 0 TO #source[]-1
                IF In(source[i], target[], attribDef, 0) < 0 THEN
                    RETURN 0 //sets are not equal
                ENDIF
            ENDFOR
            RETURN 1 //sets are equal
        ENDIF
    CASE bag:
        GetElements(s, source[]); GetElements(t, target[]);
        IF (#source[] == #target[]) THEN
            FOR i = 0 TO #target[]-1
                IF (p = IN(target[i], source[], attribDef, 0)) < 0 THEN
                    RETURN 0 //bags not equal
                ELSE
                    Remove source[p] from source[]
                ENDIF
            ENDFOR
            RETURN 1 //bags are equal
        ENDIF
    ENDSWITCH
    RETURN 0
}
```

An element of `s` is in `t`, they must be equal. If `s` and `t` are bags, if they have the same number of elements and each element of `t` is in `s` and they have the same number of all repeated elements, then `s` and `t` must be equal. This function calls `In` to see if each element of `t` is
in $s$ and removes the same element from $s$ to guarantee that they have the same number of all repeated elements.

2.5.5.4. Manipulate Values

Three functions (GetElements, MatchParenthesis and MatchQuote) are used to manipulate the value of an attribute. They are discussed in this section.

2.5.5.4.1. Function GetElements

Function GetElements is used to put all the elements of an aggregate value into array $elements[]$. Parameter value is the string containing the aggregate value. For example, if value is a set of set of strings, it looks like:

$((\text{'abcd'},\text{'efg'}),(\text{'hijk'},\text{'lmn'}))$

After GetElements is called, the result in $elements[]$ would look like:

$elements[0] = (\text{'abcd'},\text{'efg'})$

$elements[1] = (\text{'hijk'},\text{'lmn'})$

The algorithm of function GetElements is shown in Figure 2-21.

GetElements first removes the outermost parentheses from value. Now value is:

$(\text{'abcd'},\text{'efg'}),(\text{'hijk'},\text{'lmn'})$

Then this function executes a WHILE loop to get every element out of value and insert it into $elements[]$. If the first character is an opening parenthesis, this function calls MatchParenthesis to find the position of the corresponding closing parenthesis, gets this element out from value and puts it into $elements[]$. If the first character is an opening
single quote, this function calls \textit{MatchQuote} to find the position of the corresponding closing single quote, gets this element out from \textit{value} and puts it into \textit{elements[]}. If \textit{value} is neither an aggregate of aggregate value nor an aggregate of strings, it must be an aggregate of other simple value like real, integer, logic, etc. In this case, this function finds the position of the next comma, the string before this comma is the element wanted.

\begin{verbatim}
GetElements(value, elements[])
{
    i = 0
    value = substr(value, 2, length(value)-2)
    WHILE value != "" DO
        SWITCH value[0]
            CASE ( : // Open parenthesis
                j = MatchParenthesis(value, 1)
            CASE ' : // Quote
                j = MatchQuote(value, 1)
            DEFAULT:
                j = index(value, ",") - 1
                IF j==0 THEN
                    j = length(value)
                ENDIF
        END SWITCH
        elements[i] = substr(value, 1, j)
        i++;
        IF j+1 < length(value) THEN
            value = substr(value, j+2)
        ELSE
            value = ""
        ENDIF
    ENDWHILE
}
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{algorithm.png}
\caption{Algorithm of Function \textit{GetElements}}
\end{figure}

2.5.5.4.2. Function \textit{MatchParenthesis} and \textit{MatchQuote}

Function \textit{MatchParenthesis} and \textit{MatchQuote} are called by \textit{GetElements} to find the matched closing parenthesis or single quote. Their algorithms are shown in Figure 2-22.
MatchParenthesis(value, Position) {
    Switch value[Position+1]
    CASE { :
        DO
            Position = MatchParenthesis(value, Position+1) + 1
        WHILE value[Position] == "",
    CASE ' :
        DO
            Position = MatchQuote(value, Position+1) + 1
        WHILE value[Position] == "",
    DEFFFFLT:
        Position = Index( value[Position--end of value], "" );
    ENDSWITCH
    RETURN Position
}

MatchQuote(value, Position) {
    p = index( substr(value, position+l), " , , ");
    q = index( substr(value, position+l), " , ");
    IF(p == q)
        RETURN MatchQuote(value, p+l) ;
    ELSE
        RETURN q+Position;
    ENDIF
}

Figure 2-22 Algorithms of Function MatchParenthesis and MatchQuote

2.5.5.4.2.1. Parameters

value is the string containing the aggregate value like in function GetElements.

position is an index integer pointing to the position of the opening parenthesis or single quote in value whose matched closing parenthesis or single quote is needed.

2.5.5.4.2.2. Description

There are two special cases to consider in MatchParenthesis. One is that value is an aggregate of aggregate value like set of set of integers:

((1,3),(1,4,5),(6,7,8))
and \textit{position} is pointing to the first opening parenthesis, then the next one is still an opening parenthesis. \textit{MatchParenthesis} is called repeatedly until it reaches the one matched. In the above example, \textit{MatchParenthesis} is called three extra times to find the closing parenthesis after 3, 5 and 8 respectively.

Another special case of \textit{value} is an aggregate of strings like:

\[
('abcdefg', 'hijklmn', 'opq')
\]

and a closing parenthesis might be in any of these strings. Similarly, \textit{MatchQuote} is called three times to find the last closing parenthesis.

A special case in function \textit{MatchQuote} is that a string may have a single quote. E.g., if \textit{value} is an aggregate of strings like:

\[
('abcdefg''hijklmn', 'opqrst')
\]

and \textit{position} is pointing to the quote before \textit{a}, the corresponding one should be the one after \textit{n}. Notice that if a string contains a single quote, it needs another quote preceding it. In this case, this function searches for the first quote position of two continuous quotes and the position of a single quote starting from \textit{a} in the above string. If they are the same, this function passes the two continuous quotes and calls \textit{MatchQuote} again to find the next one which is wanted.
2.5.6. Merge Relations (CompleteRelation)

Function CompleteRelation is called at the end of function MergeDatabase to merge each non-defining relation for corresponding source and target instances. Figure 2-23 presents its algorithm.

```plaintext
CompleteRelation (entityName, attribName, cardinality, relationMergeRule)
{
    QUERY source FOR S[] = {all the OIDs that are of type entityName}
    QUERY source FOR VS[] = {values of attribName from all elements in S}
    FOR i=0 TO #S[] DO
        targetOID = MapTarget(S[i])
        vt = {MapTarget(OID): OID IN VS[i]}
        SWITCH (cardinality)
        case .singleRef.
            SET attribName of targetOID TO vt
        case .setRef.
            SWITCH (relationMergeRule)
            case .union.
                FOR EACH t IN vt DO
                    ADD t TO attribName OF targetOID IF t DOES NOT ALREADY EXIST
                ENDFOR
            case .source.
                SET attribName of TargetOID to vt
            default
                Error // relationMergeRule must be .source. or .union.
       ENDSWITCH
    ENDFOR
}
```

Figure 2-23 Algorithm of Function CompleteRelation

2.5.6.1. Parameters

Parameter entityName is the name of the entity whose relation is to be merged. The other three parameters are the attributes of the relation that belongs to entityName.

2.5.6.2. Description

Function CompleteRelation first gets all the OIDs in source that are of type entityName (stored in S[]) and selects all the values of the non-defining relation from
these instances (stored in $VS[j]$). Then for each element in $S[j]$, this function gets its corresponding $targetOID$ and the corresponding value $vt$ of $VS[i]$. Now this function needs to see if this relation is an aggregate value. If not, it sets $targetOID$ to have the same mapped relation value $vt$. Otherwise, if the value of $relationMergeRule$ is union, this function adds the extra pointers in $vt$ to the value of the relation in $targetOID$. If the value of $relationMergeRule$ is source, it sets $targetOID$ to have the same mapped relation value $vt$.

2.6. Target Database Information

In section 2.5, several functions like $dbDelete$ and $dbShift$ are called to manipulate the data in target database like deleting an entity instance from the target or shifting the inheritance level of an entity instance in the target. These functions are explained in the following section.

A fully implemented OODBMS should be able to provide the function for deleting an entity instance and updating all the references that reference this entity instance in one step. But the EQL query language can not delete the data entity instance and automatically update the pointers that point to the deleted entity instance. So, the functions described in the next section are designed to remedy this defect.

Like the merge rules set up for internal merge, a set of rules for the target database manipulation needs to be constructed before developing a set of functions for operating the target database. The merge rules can not be used since the target represents the local tool’s data entirely. It has attributes that are not modified as a result of the merge, but
returned back into the new target directly. These attributes are meant to output the internal tool’s view of the manufacturing information, but cannot be updated by the outside view.

Function \textit{dbDelete} and \textit{dbShift} need to know about all attributes and relations to work correctly. Figure 2-24 is the EXPRESS schema for the information that \textit{dbDelete} and \textit{dbShift} need. This schema is much simpler than the schema of the internal merge rules since it only needs to contain all the information of the target database structure.

\begin{verbatim}
ENTITY EntityDef
  ABSTRACT SUPERTYPE OF (ONEOF (RootDef, ChildDef));
  name : STRING;
  attrs : SET OF STRING;
  references : SET OF EntityReference;
  children : SET OF ChildDef;
  definingAttrs: SET OF DefiningAttrib;
  UNIQUE
    key : name;
END_ENTITY;

ENTITY RootDef
  SUBTYPE OF (EntityDef);
END_ENTITY;

ENTITY ChildDef
  SUBTYPE OF (EntityDef);
  parent : EntityDef;
END_ENTITY;

ENTITY EntityReference;
  entityDef : EntityDef;
  attribName : STRING;
  attribCardinality: cardinality;
  identifying : BOOLEAN;
END_ENTITY;

TYPE cardinality = ENUMERATION OF (singleRef, setRef, listRef, arrayRef, bagRef);
END_TYPE;

ENTITY DefiningAttrib;
  name : STRING;
  aggregateLevel: INTEGER;
END ENTITY;
\end{verbatim}

Figure 2-24 the EXPRESS Schema for Target Database Manipulation
Every entity instance in the target can be a RootDef, which is the highest level parent or non-inheritance entity, or a ChildDef, which is an inheritance of other entities. There are five attributes of EntityDef, which is the super type of RootDef and ChildDef:

- **name**— the name of the entity.
- **attrs**— all the names of the attributes whose data is contained in the entity. They include all the names of the dependencies that this entity contains, the pointer attributes, relations, the key attributes and the information attributes.
- **references**— all other entities that contain pointers pointing to this entity. This kind of reference can be relation or pointer attribute.
- **children**— points to all child entities (ChildRef) of this entity.
- **definingAttribs**— all pointer attributes of this entity.

Entity ChildDef has an extra attribute— *parent* which points to the parent of this child entity. Entity EntityReference is the description of the *references* attribute of entity EntityDef. It has three attributes:

- **entityDefinition**— points to the entity which contains the pointer
- **attributeName**— name of the attribute which contains the pointer
- **attributeCardinality**— cardinality of the pointer attribute
- **identifying**— indicates if this reference is a dependent relation

Entity DefiningAttrib is the description of the *definingAttribs* attribute of entity EntityDef. It has two attributes:
2.7. Functions for Target Database Manipulation

Given the above explanation of the target database information schema, function \textit{dbDelete} and \textit{dbShift} can be implemented. They are called by the internal merge functions in section 2.5 and need to call several other functions to finish the data manipulation. All these functions are presented in the following sub-sections hierarchically.

2.7.1. Manipulate the Target

Two highest level functions for manipulated the target database — \textit{dbDelete} and \textit{dbShift} — are discussed in this section.

2.7.1.1. Function \textit{dbDelete}

Figure 2-25 presents the algorithms of function \textit{dbDelete} and \textit{dbShift}. Function \textit{dbDelete} is called by some internal merge functions to delete an entity instance and all the references that point to this entity instance in the target database.
dbDelete(schema, dbase, OID, entityName) 
{
    IF entityName IS NULL THEN
        entityName = dbGetRootName(schema, dbase, OID)
    ENDIF
    RootDef = dbGetRootDef(schema, entityName)
    dbUpdateReferences(schema, dbase, OID, "NULL", RootDef);
    A = RootDef.definingAttribs
    c = dbFindChild(schema, dbase, OID, RootDef)
    WHILE c!=NULL DO
        A = A UNION c.definingAttribs
        c = dbFindChild(schema, dbase, OID, c)
    ENDWHILE
    FOR EACH a IN A DO
        GetOIDs(dbase, OID, a.name, a.aggregateLevel, 6elements[])
        FOR i=0 TO #elements[]-1 DO
            dbDelete(schema, dbase, elements[i], "NULL")
        ENDFOR
    ENDFOR
    DELETE OID from dbase
} 

dbShift(schema, dbase, OID, EntityName)
{
    newOID = CREATE a new entity of type EntityName in dbase
    rootDef = dbGetRootDef(schema, EntityName)
    dbCopyAttribs(schema, dbase, OID, newOID, rootDef)
    dbUpdateReferences(schema, dbase, OID, newOID, rootDef)
    DELETE OID from dbase
    RETURN newOID
} 

Figure 2-25 Algorithms of Function dbDelete and dbShift

2.7.1.1.1. Parameters

dbDelete has four parameters. Parameter schema is the data file that holds the target schema information. dbase is the data file of the target database. OID is the entity instance to be deleted from the target. entityName is the name of the entity to be deleted.

2.7.1.1.2. Description

Since there may be some entity instances pointing to OID, all these pointers need to be deleted before the OID instance is deleted. Function dbUpdateReferences is called to
delete all the pointers that point to OID and one of its parameters—the root entity instance id (in the target information data file) of OID is returned by function dbGetRootDef. If the value of entityType is not provided when other functions call dbDelete, function dbGetRootName is called to get it since dbGetRootDef needs it.

OID may also have some pointer attributes. In order to delete them, function dbDelete has to delete all the entity instances that they point to. The second block in this function collects all the pointer attributes (definingAttribs) that belong to OID and stores them in A. Then for each pointer attribute in A, GetOIDS is called to get all the pointer entity instances (in the target information data file) that are pointed to by this pointer attribute and to put them into elements[]. Another FOR loop is used to call dbDelete recursively and delete all the elements in elements[]. After these, all the pointer entity instances that are pointed to by OID are deleted. In the end, OID is deleted.

2.7.1.2. Function dbShift

Function dbShift is used to shift an entity instance to another inheritance level in the same inheritance tree.

The first three parameters of function dbShift are the same as function dbDelete. The fourth parameter—EntityName is the name of the entity that OID is to be shifted to.

A new entity instance newOID of type EntityName is created in the target database followed by copying all the common attributes from OID to newOID by calling dbCopyAttribs. Then all the pointers that point to OID are changed to newOID by calling dbUpdateReferences and instance OID is deleted at the end.
2.7.2. Change References or Attributes

Function `dbUpdateReferences` and `dbCopyAttrs`, which are called by the above functions, are explained in this section.

2.7.2.1. Function `dbUpdateReferences`

Function `dbUpdateReferences` is called to delete all the pointers that point to an entity instance or change these pointers to point to another entity instance. Figure 2-26 presents the algorithm for this function.

```
dbUpdateReferences(schema, dbase, OID, newOID, RootDef)
{
    references = QUERY schema for RootDef.references
    c = dbFindChild(schema, dbase, OID, RootDef)

    IF (newOID != NULL) THEN
        newc = dbFindChild(schema, dbase, newOID, RootDef)
        WHILE (c == newc) AND (c IS NOT NULL)
            R = QUERY schema for c.references
            references = references UNION R
            c = dbFindChild(schema, dbase, OID, c)
            newc = dbFindChild(schema, dbase, newOID, newc)
        ENDWHILE
        FOR EACH r IN references DO
            UPDATE all instance of entities of type r.entityDefinition
            WHERE r.attributeName CONTAINS OID
                (DELETE OID FROM r.attributeName AND ADD newOID TO
                r.attributeName) IN dbase
        ENDFOR
        references = {}
    ENDIF

    WHILE (c != NULL)
        R = QUERY schema for c.references
        references = references UNION R
        c = dbFindChild(schema, dbase, OID, c)
    ENDWHILE
    FOR EACH r IN references DO
        IF r.cardinality == .singleRef. AND r.identifying == .T. THEN
            DELETE all the instances that reference to 010
        ELSE
            UPDATE all instance of entities of type r.entityDefinition
                WHERE r.attributeName CONTAINS OID
                (Delete OID from r.attributeName) IN dbase
        ENDIF
    ENDIF
}
```

Figure 2-26 Algorithm of Function `dbUpdateReferences`
2.7.2.1.1. Parameters

There are five parameters for this function: `schema` and `dbase` are the same as in `dbDelete`, `newOID` is the entity instance substituting `OID` in all the relations that currently reference to `OID`, and `RootDef` is the root entity instance id (from the target information data file) of `OID` and `newOID`. `OID` and `newOID` must be in the same inheritance tree.

2.7.2.1.2. Description

If `newOID` is NULL, all the pointers that reference `OID` are deleted, else, all the common references that can reference both `OID` and `newOID` are collected and changed to reference `newOID`. All the extra references that only reference `OID` are deleted.

Local variable `references` is used to collect all the common references of `OID` and `newOID` in the first part of the algorithm. Its original value is set to all the references of the root entity of `OID` and `newOID`. Starting from the `RootDef`, this function calls `dbFindChild` to find the next entity level in the `RootDef` to `OID` inheritance path, stores it in `c`, and finds the next entity level in the `RootDef` to `newOID` inheritance path and stores it into `newc`. IF `c` equals `newc`, this function collects their common references and goes one level down by calling `dbFindChild` again and compares them. After the first `WHILE` loop, all the common references of `OID` and `newOID` should be collected. Then for each of these references, this function changes it to reference `newOID` instead of `OID`.

The second `WHILE` loop is used to collect all the extra references that only reference `OID`. All of these references are deleted in the last query. If a reference is a single
pointer and it represents a dependent relation, all the instances that reference to \( OID \) should also be deleted.

If \( newOID \) is NULL, this function goes to the second \( WHILE \) loop directly and delete all the references to \( OID \).

2.7.2.2. Function \( dbCopyAttribs \)

Function \( dbCopyAttribs \) is used to copy the values of all the common attributes of \( OID \) and \( newOID \), from \( OID \) to \( newOID \), and delete all the extra pointer attributes that belong to \( OID \) only. Its algorithm is shown in Figure 2-27. All the parameters of this function are the same as \( dbUpdateReferences \).

```
function dbCopyAttribs(schema, database, oid, newOID, RootDef)
    attributes = query schema for RootDef.attribs
    c = dbFindChild(schema, database, oid, RootDef)
    newc = dbFindChild(schema, database, newOID, RootDef)
    while (c == newc) and (c is not NULL)
        A = query schema for c.attribs
        attributes = attributes union A
        c = dbFindChild(schema, database, oid, c)
        newc = dbFindChild(schema, database, newOID, newc)
    endwhile
    V = query database for all values of attributes A of oid
    update instance newOID (set {A} = {V}) in database
    while (c != NULL) do
        P = c.definingAttribs
        for each p in P do
            GetOIDs(oid, p.name, p.aggregateLevel, &elements[])
            for i = 0 to #elements[] - 1 do
                dbDelete(schema, database, elements[i], "NULL")
            endfor
        endfor
        c = dbFindChild(schema, database, oid, c)
    endwhile
endfunction
```

Figure 2-27 Algorithm of Function \( dbCopyAttribs \)
This algorithm is similar to function \textit{dbUpdateReferences}. It collects all the common attributes of \textit{newOID} and \textit{OID} by calling function \textit{dbFindChild} and comparing each level in their inheritance path. After the first \textit{WHILE} loop, it queries the values of the common attributes from \textit{OID} and sets \textit{newOID} with the same values.

The second \textit{WHILE} loop is used to delete all the extra pointer attributes that belong to \textit{OID} only.

\textbf{2.7.3. Other Functions}

All the other functions that are called by the previous functions are discussed in this section. They are \textit{dbGetRootName}, \textit{dbGetRootDef}, \textit{dbFindChild}, and \textit{GetOIDs}.

\textbf{2.7.3.1. Function \textit{dbGetRootName} and \textit{dbGetRootDef}}

Function \textit{dbGetRootName} and \textit{dbGetRootDef} are used to find the name and instance id (from the target information data file) of the root entity of an entity instance in the target. Figure 2-28 presents the algorithms of these two functions.

In function \textit{dbGetRootName}, parameter \textit{schema} and \textit{dbase} have the same meaning as in previous functions. \textit{OID} is the object id of the entity instance in the target whose root entity name is wanted. This function just collects all the root entities and checks if any one of them is the root of \textit{OID}. It returns the name of the found root entity or NULL if it fails.

In function \textit{dbGetRootDef}, parameter \textit{schema} is the same as in \textit{dbGetRootName}. Parameter \textit{name} is the name of the entity whose root entity instance id is wanted. This
function goes to the upper level entity of \textit{name} each time until it finds the root and returns its name.

function goes to the upper level entity of \textit{name} each time until it finds the root and returns its name.

\begin{verbatim}
dbGetRootName(schema, dbase, OID)
{
    E = {names of all RootDef entities in dbase}
    FOR EACH e IN E DO
        IF OID IS a type of e THEN
            RETURN e.name
        ENDIF
    ENDFOR
    RETURN NULL
}
dbGetRootDef(schema, name)
{
    IF name Exists in schema THEN
        While (name is a ChildDef in schema)
            Name = Name.Parent
        ENDWHILE
        RETURN Name
    ELSE
        Error: Entity is not in schema Definition
    ENDF
}
\end{verbatim}

\textbf{Figure 2-28 Algorithms of Function \textit{dbGetRootName} and \textit{dbGetRootDef}}

\subsection*{2.7.3.2. Function \textit{dbFindChild}}

Function \textit{dbFindChild} is used to find the child of an entity (the last parameter), which is still a parent of another entity (the third parameter) or this entity itself. For example, if entity \textit{GraduateStudent} inherits from entity \textit{Student} and entity \textit{Student} inherits from entity \textit{Person}. If parameter \textit{EntityDef} points to entity \textit{Person} in the data file of the target information schema and \textit{OID} points to an entity instance of \textit{GraduateStudent} in the target. This function returns the instance id of entity \textit{Student} from the target information data file, which is one level down the inheritance path from \textit{Person} to \textit{GraduateStudent}. 
Figure 2-29 presents this function’s algorithm. Parameter \textit{schema} and \textit{dbase} are the same as in previous functions. All the children of \textit{EntityDef} are selected first. Then for each of them, this function checks if \textit{OID} belongs to this type, if so, it returns this child’s instance id. This function returns NULL after it runs through every child and fails to find one.

\begin{verbatim}
    dbFindChild(schema, dbase, OID, EntityDef)
    { 
        C = [EntityDef.children in schema]
        IF (C is EMPTY)
            RETURN NULL;
        ENDF
        FOR each c in C
            IF (OID is a type of c.name in dbase)
                return c;
            ENDF
        ENDF
        RETURN NULL
    }
\end{verbatim}

*Figure 2-29 Algorithm of Function \textit{dbFindChild}*

2.7.3.3. Function \textit{GetOIDs}

Function \textit{GetOIDs} (Figure 2-30) is used to get all the OIDs of the pointer entity instances that are pointed to by a pointer attribute of another entity instance and store them in \textit{elements[i]}. \textit{name} is the name of the attribute in \textit{oid}. \textit{level} is the aggregate level of this pointer attribute.

\begin{verbatim}
    GetOIDs(dbase, oid, name, level, elements[])
    { 
        FOR (i=0; i<level; i++)
            name |= "[]"
        ENDF
        Eql_exec("SELECT name from OID", elements[])
    }
\end{verbatim}

*Figure 2-30 Algorithm of Function \textit{GetOIDs}*
Function *GetOIDs* just appends number *level* of brackets to *name* and runs a query to select all the pointer entity instances from the target.
Chapter 3

Generic Examples of the Internal Merge

3.1. Introduction

In this chapter, four examples are presented that show how the internal merge works with all the situations designed in the pseudo code of chapter 2. A simple example of each case is given in the following sections.

3.2. Merge Inheriting Entities

The schema of the example of this case is shown in Figure 3-1. Entity Person is the parent of Student and Employee. Student is the parent of UnderGrad (undergraduate student) and Grad (graduate student). Employee is the parent of Faculty. In the merge rules data file of this example, entity Person is an instance of IndependentEntityRule and all the other entities are instances of ChildEntityRule since they are the children of Person. The key attribute of all of these entities is ssNumber (social security number). Attribute students of entity Faculty is a set of pointers that point to entity Student. This is a non-identifying relation between these two entities. All the other attributes in the schema are information attributes.
Figure 3-1 Schema of Inheriting Example

Figure 3-2 presents an example of the target and source data file based on the schema in Figure 3-1. The Part21 file of the source data is on the top. The Part21 file of the original target data is at the bottom.

```
#10=Person('Tom', '111-11-1111');
#20=Grad('John', '222-22-2222', 'Senior', 3.80E+00, 2300);
#30=Student('Alex', '333-33-3333', 'Freshman', 3.50E+00);
#40=Grad('Lynn', '444-44-4444', 'Junior', 3.70E+00, 2200);
#50=Faculty('Judd', '555-55-5555', 'Professor', 8.0E+04, (#20, #30, #40));
```

```
#10=Person('Tom', '111-11-1111');
#20=Grad('John', '222-22-2222', 'Senior', 3.80E+00);
#30=Student('Alex', '333-33-3333', 'Freshman', 3.50E+00);
#40=Grad('Lynn', '444-44-4444', 'Junior', 3.70E+00, 2200);
#50=Faculty('Judd', '555-55-5555', 'Professor', 8.0E+04, (#20, #30));
```
If the values of instanceMergeRule and inheritanceMergeRule are union, the internal merge begins with Person ‘Tom’ in the source, compares its key value ‘111-11-1111’ with each Person entity instance in the target by calling function Find and it finds that the first instance in the target is the matched one. Then it merges their information attributes by calling function MergeInfoAttribs, which is name (‘Tom’), since they are in the same inheritance level.

Grad ‘John’ is picked next from the source. There is a corresponding Student ‘John’ in the target. The internal merge algorithm needs to bring Student ‘John’ instance in the target down to the Grad level by calling function dbShift and merges it with the matched instance in the source. It first copies all the attributes of Student ‘John’ to a new Grad instance in the target by calling function dbCopyAttribs and replaces all the references of Student ‘John’ with the new one by calling function dbUpdateReferences. In this case, Faculty ‘Judd’ in the target is referencing Student ‘John’ and this pointer needs to be changed to reference Grad ‘John’. After this reference is updated, the original Student ‘John’ can be deleted from the target. Then, the new instance is merged with the Grad ‘John’ in the source and the GRE score is added to this instance. After this step, the data in the target is shown in Figure 3-3.

```plaintext
#10=Person('Tom', '111-11-1111');
#20=Grad('Alex', '333-33-3333', 'Freshman', 3.50E+00, 2000);
#30=Faculty('Judd', '555-55-5555', 'Professor', 8.0E+04, (#20, #40));
#40=Grad('John', '222-22-2222', 'Senior', 3.80E+00, 2300 );
```

Figure 3-3 the Target Data
The internal merge algorithm then picks Student ‘Alex’ from the source, and finds out there is a corresponding Grad ‘Alex’ in the target. Since Grad entity is lower than Student in the inheritance tree, it contains more information than Student. Only the common attributes of Student and Grad are merged and the extra attribute GRE score in Grad stays the same. After this step, the target is the same as in Figure 3-3.

For Grad ‘Lynn’ in the source, function Find of internal merge fails and function CreateEntity is called to create a copy of Grad ‘Lynn’ into the target. After this step, the target data is shown in Figure 3-4. Notice that the relation between Faculty ‘Judd’ and Grad ‘Lynn’ has not been copied yet. This is done by calling function CompleteRelation in the last step of internal merge.

```
#10=Person('Tom', '111-11-1111');
#20=Grad('Alex', '333-33-3333', 'Freshman', 3.50E+00, 2000);
#30=Faculty('Judd', '555-55-5555', 'Professor', 8.0E+04, (#20, #40));
#40=Grad('John', '222-22-2222', 'Senior', 3.80E+00, 2300);
#50=Grad('Lynn', '444-44-4444', 'Junior', 3.70E+00, 2200);
```

**Figure 3-4 the Target Data**

Faculty ‘Judd’ is picked next from the source and merged with the corresponding instance in the target. The relation of it is not merged. So after this step, the target stays the same as in Figure 3-4.

At last, Function CompleteRelation is called to merge all the non-identifying relations between the corresponding target and source instances. There is only one relation in this example that needs to be merged, which belongs to entity Faculty. The final target database is presented in Figure 3-5.
If the values of instanceMergeRule and inheritanceMergeRule are source, the internal merge picks every instance from the source and shifts the corresponding target instance to the same inheritance level and merges them. Then it deletes all the target instances that do not have corresponding source instances. The final target database is presented in Figure 3-6. Student ‘John’ and Grad ‘Alex’ in the original target are shifted to Grad ‘John’ and Student ‘Alex’ respectively, that are at the same inheritance level as their corresponding source instances. Grad ‘Lynn’ is copied from the source since it does not have a corresponding instance in the original target. There is no extra target instance that needs to be deleted since all the original target instances have matched source instances.

3.3. Check and Create Dependent Relations

The schema of an example of dependent relations is presented in Figure 3-7. Entity ClassOffering and Student are two independent entities. ClassOffering gives the information of the offered classes. It is identified by its title. Student gives the
information of a student. It is identified by its id. There are two relations in the above schema: \textit{ClassOffering} points to a set of \textit{Enrollments} and \textit{Enrollment} points to a \textit{Student}. By using these two relations, we can know how many students select a class and what their names and grades are. An \textit{Enrollment} instance can only be identified by knowing the class selected by a given student. So these two relations are defining relations for entity \textit{Enrollment}.

An example of merging data of this schema is presented in Figure 3-8. The data of the source is on the top and that of the target is at the bottom. The following discussion is based on that the value of \textit{instanceMergeRule} in the merge rules data file is union.

The internal merge algorithm merges all the independent entities followed by dependent entities. First, all the instances of \textit{ClassOffering} or \textit{Student} are selected from the source and merged with the corresponding instances in the target. The target data stays the same as in Figure 3-8 after this step.
All the instances of dependent entity *Enrollment* are merged next. For the first instance (#60) in the source, *Find* is called to find its matching instance in the target. It gets all the target instances of *Enrollment* and compares each one of them with #60 in the source until it finds a match. First it compares #60 from the source with #6000 from the target. Since *Enrollment* has two dependent relations, all of them are checked in function *CheckDependencies*. This function picks one dependent relation of instance #60 in the source, which is pointing from *ClassOffering* to *Enrollment*, to check if it is the same for the source and target instances. It first needs to see if the defining entity contains the pointer, which is true in this case. Then it queries the source to find one *ClassOffering* instance that points to #60 instance and gets #10. It does the same thing for #6000 in the target and gets #1000. Since #10 from the source corresponds to #1000 from the target, *CheckDependencies* checks the next dependent relation, which points from *Enrollment* to *Student*. This is an easier case, since dependent entity contains the pointer. It gets the
value of this pointer from #60 in the source, which is #30, and compares this with the one from #6000 in the target, which is #3000. #30 in the source corresponds to #3000 in the target. So #60 in the source corresponds to #6000 in the target after comparing all their dependent relations. Their infoAttribs will be merged finally. After these, the target data stays the same as in Figure 3-8.

For the next instance of Enrollment in the source—#70, after the same checking process, no corresponding instance is found in the target. So function CreateEntity is called to create a new one into the target. This function creates an entity in the target with all the infoAttribs from the source instance #70 and then updates its defining relations based upon whether the pointer is in the defining entity or the dependent entity. For the dependent relation pointing from ClassOffering to Enrollment, this function queries the ClassOffering entity instance from the source, which points to #70, and gets #10. Then this function finds its corresponding instance from the OID map, which is #1000 in the target and updates the pointer of the #1000 instance in the target to include the created target instance. For the other dependent relation, which points from Enrollment to Student, this function queries the value of this pointer from the #70 instance in the source, which is #40, finds its corresponding instance from the OID map, which is #4000 in the target, and updates the pointer of the created target instance to point to #4000. The result of the target data is shown in Figure 3-9 after this step. #9000 is the new added entity instance. #1000 points to it and it points to #4000.
For #80 and #90 in the source, *Find* returns #7000 and #8000 in the target respectively after the same checking process and merges them. The target data is identical to Figure 3-9 after this step.

For #100 in the source, *Find* fails and *CreateEntity* is called again. The result is presented in Figure 3-10. #9100 is the new added entity instance. #2000 points to it and it points to #5000. Figure 3-10 is also the final result of this example.

If the value of *instanceMergeRule* in the merge rules data file is source, the final result of the internal merge is still the same as in Figure 3-10 since there is no extra target instance that does not have corresponding source instance and needs to be deleted.

### 3.4. Merge Aggregate Values

The schema of this example is presented in Figure 3-11.
In the merge rules data file, entity A is an IndependentEntityRule. Attribute name is the key attribute for entity A, which has four other aggregate information attributes. The data files of the source and target are presented in Figure 3-12. The source file is on the top and the target file is at the bottom.

Instance #10 from the source is merged with instance #100 from the target since they have the identical key attribute value ‘Instance1’. Function MergeInfoAttrs is called next to merge their four infoAttribs. For each aggregate infoAttrib, if aggregateMergeRule is union, function MergeAggreValue is called to merge its values from the source and target instances. The following explanation is for an union. For a source, the aggregate attribute value is just copied from the source to target.
Value (1,2,3) from the source instance is merged with value (3,4,2) from the target instance. These values are sets of integers. Function MergeAggreValue picks each element in (3,4,2) and calls function In to see if it is in (1,2,3). 3 is selected first and passed to In, which calls Equal to compare 3 with every one in (1,2,3) until it finds an equal one. Then 3 is moved out from (1,2,3) which becomes (1,2). The second element 4 of (3,4,2) is selected next and compared with each one in (1,2) and nothing needs to be removed. The third element 2 of (3,4,2) is selected next and compared with each one in (1,2) and 2 is removed. (1,2) becomes (1). Now MergeAggreValue joins (1) and (3,4,2), to produce (1,3,4,2), which is the merged value.

Value ('cylinder', 'cube', 'cone') from the source instance is merged with value ('cylinder', 'cube', 'cone') from the target instance next. These values are arrays of strings. They must be identical. MergeAggreValue just checks if they are equal by calling Equal. If not, an error is generated and the internal merge stops.

Value «'dog', 'deer'), ('crow', 'sparrow')) from the source instance is merged with value «'dog', 'cat'), ('crow', 'sparrow')) from the target instance next. These values are lists of sets of strings. The first element ('dog', 'cat') is selected from the target value and In is called to see if it is in the source value. It is not, MergeAggreValue stores it in the merged value. The second element ('crow', 'sparrow') is selected from the target value and In is called again. This time it is in the source value and the index of its position in the source value is returned. Then, all the elements from 0 to the index in the source value are moved out and appended to the end of the merged value. Now the final merged value is ((‘dog’, ‘cat’),(‘dog’, ‘deer’), (‘crow’, ‘sparrow’)).
Value ((1.0, 2.0), (1.0, 2.0), (3.0, 4.0), (1.0, 3.0)) from the source instance is merged with value ((4.0, 2.0), (4.0, 3.0), (2.0, 1.0)) from the target instance next. These values are bags of sets of reals. The first element in the target value -- (4.0, 2.0) is selected and \textit{In} is called to see if it is in the source value. It is not. So, it is put into the merged value -- ((4.0, 2.0)). The second element in the target value -- (4.0, 3.0) is selected and it is in the source value. So it is deleted from the source and appended to the merged value. Now the merged value is ((4.0, 2.0), (4.0, 3.0)). The source value is ((1.0, 2.0),(1.0, 2.0),(1.0, 3.0)). The third element in the target value -- (2.0, 1.0) is selected and it is in the source. It is deleted from the source and appended to the merged value. Now the merged value is ((4.0, 2.0), (4.0, 3.0), (2.0, 1.0)). The source value is ((1.0,2.0),(1.0,3.0)). The final merged value is the union of these two values-- ((4.0, 2.0), (4.0, 3.0), (2.0, 1.0), (1.0,2.0), (1.0, 3.0)).

The final internal merge result is presented in Figure 3-13.

![Figure 3-13 The Final Result](image)

3.5. Merge Pointer Attributes

The schema of this example is presented in Figure 3-14.
Attribute `attrib1` of entity `A` is represented by a set of entity `B` instances and attribute `attrib2` of entity `A` is represented by a set of entity `C` instances. Similarly, attribute `attrib` of entity `B` is represented by a set of entity `C` instances.

The data files of the source and target are presented in Figure 3-15. The source file is on the top and the target file is at the bottom.
In the merge rules data file, entity $A$ is an IndependentEntityRule instance. Entity $B$ and $C$ are AttribEntityRule instances. Attribute name is the key attribute for each of these three EntityRules. All the other attributes are information attributes. $attrib1$, $attrib2$ and $attrib$ are aggregate of PointerRules. The following discussion is based on that the values of aggregateMergeRule of all the aggregate attributes are union.

The internal merge algorithm starts by merging all the IndependentEntityRule ($A$) instances. So instance #100 ('A1') is selected from the source and merged with the target. Function $Find$ is called and its corresponding target instance #10 is returned since they have the same key attribute value – 'A1'. Then in function $MergeEntityInstance$, function $MergeInfoAttribs$ is called to merge the information attributes of these two instances, which are $attrib1$ and $attrib2$, in this case. When $MergeInfoAttribs$ finds out that attribute $attrib1$ is really an aggregate of PointerRules, $MergeAggrePointer$ is called to merge its AttribEntityRule instances and this function calls $MergeEntityInstance$ to merge the entity $B$ instance #110 from the source with #20 from the target since they have the same key attribute value – 'B1'. And function $MergeEntityInstance$ will be called again to merge #120 from the source with #30 from the target.

For merging #110 from the source with #20 from the target in function $MergeEntityInstance$, function $MergeInfoAttribs$ is called again to merge attribute value and $attrib$ of entity $B$ and $MergeEntityInstance$ is called again to merge entity $C$ instance #150 from the source with #60 from the target and #160 from the source with #70 from the target. The value of the simple attribute value is copied from the #110 in the source
to #20 in the target. After this step, the target appears as in Figure 3-16. Compared to the original target, the values in #20, #60 and #70 were updated.

\[
\begin{align*}
\#10 &= \text{A('A1', (#20, #30), (#40, #50))}; \\
\#20 &= \text{B('B1', 9, (#60, #70))}; \\
\#30 &= \text{B('B2', 6, (#80, #90))}; \\
\#40 &= \text{C('C1', 1.15)}; \\
\#50 &= \text{C('C2', 1.25)}; \\
\#60 &= \text{C('C1', 1.11)}; \\
\#70 &= \text{C('C2', 1.22)}; \\
\#80 &= \text{C('C1', 2.33)}; \\
\#90 &= \text{C('C2', 1.44)};
\end{align*}
\]

Figure 3-16 the Target Data File

According to a similar merging process, #120 from the source is merged with #30 from the target. Then the target appears as in Figure 3-17. Compared to Figure 3-16, the values in #30 and #80 were updated.

\[
\begin{align*}
\#10 &= \text{A('A1', (#20, #30), (#40, #50))}; \\
\#20 &= \text{B('B1', 9, (#60, #70))}; \\
\#30 &= \text{B('B2', 5, (#80, #90))}; \\
\#40 &= \text{C('C1', 1.15)}; \\
\#50 &= \text{C('C2', 1.25)}; \\
\#60 &= \text{C('C1', 1.11)}; \\
\#70 &= \text{C('C2', 1.22)}; \\
\#80 &= \text{C('C1', 1.33)}; \\
\#90 &= \text{C('C2', 1.44)};
\end{align*}
\]

Figure 3-17 the Target Data File

Next, function \textit{MergeInfoAttrbs} merges attrib2 of entity A using a similar algorithm and the values in #40 and #50 are updated. After this, the target is shown in Figure 3-18 and internal merge is terminated.

\[
\begin{align*}
\#10 &= \text{A('A1', (#20, #30), (#40, #50))}; \\
\#20 &= \text{B('B1', 9, (#60, #70))}; \\
\#30 &= \text{B('B2', 5, (#80, #90))}; \\
\#40 &= \text{C('C1', 1.1)}; \\
\#50 &= \text{C('C2', 1.2)}; \\
\#60 &= \text{C('C1', 1.1)}; \\
\#70 &= \text{C('C2', 1.2)}; \\
\#80 &= \text{C('C1', 1.33)}; \\
\#90 &= \text{C('C2', 1.44)};
\end{align*}
\]

Figure 3-18 the Target Data File
If the values of the `aggregateMergeRule` of all the aggregate attributes are source in the merge rules data file. All the `AttribEntityRule` instances are deleted from the target. Then, all the corresponding `AttribEntityRule` instances are copied from the source. The final target is shown in Figure 3-19. #90 in the original target was deleted. The order of these instances is different from the source because when instance `B1` is created, all its pointer entity instances are also created.

```
#10=A('A1', (#20, #50), (#70, #80));
#20=B('B1', 9, (#30, #40));
#30=C('C1', 1.11);
#40=C('C2', 1.22);
#50=B('B2', 5, (#60));
#60=C('C1', 1.33);
#70=C('C1', 1.1);
#80=C('C2', 1.2);
```

**Figure 3-19 the Target Data File**

Note that the values of the `instanceMergeRule` and `inheritanceMergeRule` will not affect the final result, since each `IndependentEntityRule` instance in the original target has a corresponding source instance and there is no inheriting entity instance in these data files.
Chapter 4

Cost Advantage

4.1. Introduction

Cognition Corporation makes a software system called "Cost Advantage" (CA) to calculate the cost of manufacturing processes. A brief introduction of CA is given in this chapter and a description of the two translators follows. In the end, a practical example of applying the internal merge algorithm to CA is illustrated.

4.2. CA Overview

Cost Advantage is a knowledge-based software system that provides expert-level design guidance and can analyze manufacturing alternatives, producibility and predictive cost analysis. Cost Advantage consists of four components.

<table>
<thead>
<tr>
<th>Component</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CA Modeler</td>
<td>Builds or modifies existing process models.</td>
</tr>
<tr>
<td>Process Model</td>
<td>Stores knowledge of manufacturing specialists.</td>
</tr>
<tr>
<td>Cost Advantage</td>
<td>Analyzes designs using process models, including expert-level data, rules, and equations.</td>
</tr>
<tr>
<td>Note</td>
<td>Stores cost and part information.</td>
</tr>
</tbody>
</table>

Cost Advantage is a shell, which has an environment and language to develop cost models. It is fully customizable using its programming language for representing data, defining formulas and stipulating rules and restrictions.
There are two types of files at the highest level of Cost Advantage: the modeling file and the note file. The modeling file is used in constructing a cost model for an enterprise. It can represent the data, variables, structures, cost evaluation methodologies, formulas and rules. A modeling file can be specified through the window-based interface of CA, which is convenient for model developing. A note file will be generated when CA is executed according to the modeling file that is constructed by model developer. The note file is used to represent the actual cost estimations generated by users based upon the model.

The note file can be classified into two categories: component notes and assembly notes. For example, if the user want to make a bolt, which is a single part, it can be stored in a component note. But if the user want to make a part, which includes a bolt and a nut, it should be stored in an assembly note since it includes two single parts.

Figure 4-1 shows how the components in the previous table relate to each other. It also indicates some typical CA Modeler end users.

Figure 4-1 CA Modeler and Cost Advantage
Originally, a new CA model is a blank model without any data and calculating methods defined. An interface for entering all the related data needs to be designed and a model of all kinds of process and the prices for tooling and part material needs to be hierarchically constructed.

Then, when CA is running, after the user enters all the information that is necessary for cost estimation, such as the processes that they want to use to make a part, the material that they choose for the part, the size of the part, the tool that is chosen to make it, the labor rate, etc, it can produce the cost of labor, tooling, material and total cost for every operation needed to make this part.

4.3. Communication with CA

As mentioned in section 1.4, another important part of the intelligent interface to the IMDE architecture is the access-local-data modules. The design of the access-local-data modules is different for heterogeneous manufacturing databases. The communication module for one of them like CA in this particular case includes two translators: one for extracting data from the local database of CA using its report file and the other one for inserting data into the local database of CA using its design file. In this section the general formats of these two files are described and the algorithms for the two translators are explained in the next section.
4.3.1. Extract Data from CA – Report File

CA can store all the data of any note file into a report file. There are specific formats associated with this report file according to which CA inserts data.

Figure 4-2 shows an example of this report file. The first paragraph of a report file stores all the general information of the part whose cost information is reported. From the first line of Figure 4-2, one can see that this is an assembly part with name JunkAssembly. It is composed of two other parts—junk and Flange3 which are on the Reference lines. Reference indicates that parts junk and Flange3 are stored in other report files whose names are junk and Flange3.

Figure 4-2 an Example of a Report File
There are two features associated with \textit{JunkAssembly – Wire\_Brush} and \textit{Wear\_Sleeves} which are the names of the operations that are needed to make this part. All the cost information, which is in the summary window and process window of the running interface of CA, is stored in the following blocks of the report file. All the values of all the variables that were defined in the model file of each feature can also be obtained from this report file.

Thus, any information about a part can be found in the report file.

\textbf{4.3.2. Update the Local Database of CA – Design File}

Data can be inserted into the local database of CA through the design file, which has its own specific formats of different commands that are designed for different insertion functions.

Figure 4-3 is an example of a design file, which can be read by CA to generate a cost note file of a part.

\begin{verbatim}
LoadProcessModel elano9
NewNote Note1
SetNoteType Assembly
Process Assembly Path Assembly BatchSize 10.000
Feature Wire\_Brush Path Wire\_Brush
Feature Wear\_Sleeves Path Wear\_Sleeves
AddNewPart junk Parent Note1 AddFeature Wire\_Brush
SaveNote /usr/people/elano/cmedata/anyname
\end{verbatim}

Figure 4-3 an Example of a Design File
It first loads the model file named *elano9* that was constructed before. The name of the new note file is set as *Note1* and type is *Assembly*, which means that this note is for an assembly part. Two features that belong to this assembly part are added—*Wire_Brush* and *Wear_Sleeves*. *Part_junk* is added to this assembly part with feature *Wire_Brush*. In the end, this assembly part is saved as `/usr/people/elano/cmedata/anyname`.

### 4.4. Design of Two Translators

The algorithms for the two translators can be developed based on the formats of the report file and design file described in the above section. One translator is used to get data from CA and put it into a Part21 file, and the other one is used to get data from this Part21 file and insert it back to CA. In this section, the Part11 schema of the Part21 file is explained first and the illustrations of the algorithms of the two translators follow.

#### 4.4.1. CA Data Schema

Figure 4-4 presents the EXPRESS schema of the Part21 file.

Assembly and component notes of CA are represented by *PartAssembly* and *PartElement* entities respectively. Both of them inherit from the abstract entity *Part*.

Attribute *name* in *Part* is the name of this part. Attribute *features* points to a set of *Feature* entities which represent all the features of this part, such as material features and manufacturing process features. Entity *PartAssembly* can have a set of components that consist of a set of component or assembly parts. The attributes of entity *PartElement* are
the same as entity PartAssembly except for the components attribute since it represents a single component part only.

Entity Feature represents a feature of a part. FType is the name of the feature. parameters stores all the values that belong to this feature. These values can be integer,
real or text, corresponding to entity `IntegerParameter`, `RealParameter` and `TextParameter` respectively.

4.4.2. Translator for Extracting Data from CA

The algorithm of this translator is based on the format of the report file. It consists of three major functions – `ReadFile`, `ReadComponent`, and `ReadAssembly`. These three functions are explained in this section.

4.4.2.1. Function `ReadFile`

Figure 4-5 presents the pseudo code of function `ReadFile` that is first called by the `Main` function to read the report file and put the data of a part into a Part21 format file.

```plaintext
Main()
{ ReadFile(report, "NULL") }

ReadFile(report, header)
{
    IF(report != "NULL")
        OPEN the report file
    ENDF

    IF(header == "NULL")
        Name = the name of the part
        IF the first line of 'report' is for Component
            RETURN ReadComponent("NULL", Name)
        ELSE \Assembly
            RETURN ReadAssembly("NULL", Name)
        ENDF
    ELSE
        Name = the name of the part
        IF the first line of 'header' is for Component
            RETURN ReadComponent(header, Name)
        ELSE \Assembly
            RETURN ReadAssembly(header, Name)
        ENDF
    ENDF
}
```

Figure 4-5 Algorithm of Function `ReadFile`
Parameter *report* is the name of the report file generated by CA. If it is NULL, this means that the report file is already open. Parameter *header* is a string which stores the header information (the first paragraph) of the report file. If it is NULL, the header information is in the report file.

This function reads the first line of the report file or *header* to see if it is for a component or assembly part, then calls *ReadComponent* or *ReadAssembly* to get the other data. It returns the instance id that is returned by function *ReadComponent* or *ReadAssembly* for later recursive calling in *ReadAssembly*.

### 4.4.2.2. Function *ReadComponent*

Function *ReadComponent* is called by *ReadFile* to read the data of a component part from its report file and put them into a Part21 file. Figure 4-6 presents its algorithm.

Parameter *header* is the same as in function *ReadFile*. Parameter *name* is the name of the component part got from the first line of the report file.

This function gets all the data of a component report file and inserts this into the Part21 data file. It first checks if header is NULL to decide where to read the header information. There are three major data blocks in a report file of a component part: one is for all the information of the process; one is for all the information of the material; the other one is for all the information of all the features. This function goes to each data blocks, gets all the parameters and inserts them into the Part21 data file. In the end, it inserts a *PartElement* entity with its *features* attribute pointing to all the features of the three data blocks.
4.4.2.3. Function ReadAssembly

Function ReadAssembly gets all the data of an assembly report file and inserts it into the Part21 file. Figure 4-7 presents its algorithm.

The two parameters of this function are the same as in function ReadComponent. The structures of these two functions are also similar.

First, function ReadAssembly needs to decide whether to read the header information from header string or from a report file. ReferenceFiles[] is used to store all the names
of the report files that this assembly part references. \textit{NewParts[]} is used to store all the header information of every new part. \textit{ReferenceFileQTY[]} and \textit{NewPartQTY[]} are used to store the quantity of every reference part or new part.

```plaintext
ReadAssembly(header, name)
{
    IF(header == "NULL")
        FeatureNames[] = names of all the features IN report
        ReferenceFiles[] = names of all the reference files IN report
        NewParts[] = header information of every new part
    ELSE
        FeatureNames[] = names of all the features IN header
        ReferenceFiles[] = names of all the reference files IN header
        NewParts[] = header information of every new part
    ENDIF

    FILL ReferenceFileQTY[] and NewPartQTY[]

    Go to the Process block
    GET Process path and insert into DataFile an entity of 'Parameter'
    FOR EACH Characteristic IN Process block
        Get its value and insert into DataFile an entity of 'Parameter'
    ENDFOR
    INSERT INTO DataFile an entity of 'Process'

    FOR EACH Feature block of FeatureNames[]
        GET Feature path and insert into DataFile an entity of 'Parameter'
        FOR each Characteristic of this Feature block
            Get its value and insert into DataFile an entity of 'Parameter'
        ENDFOR
        INSERT INTO DataFile an entity of 'Feature' with FType = FeatureNames[i]
    ENDFOR

    FOR i=0 to #NewParts[]
        ReadFile("NULL", NewParts[i])
    ENDFOR

    FOR i=0 to #ReferenceFiles[]
        ReadFile(ReferenceFiles[i], "NULL")
    ENDFOR

    RETURN (INSERT INTO DataFile an entity of 'PartAssembly')
}
```

Figure 4-7 Algorithm of Function \textit{ReadAssembly}

There are only two data blocks in an assembly report file: one is for all the information of the process and the other one is for the information about all the features. After inserting all the parameters of these two data blocks, for each new part, \textit{ReadAssembly} calls function \textit{ReadFile} again to fill all the parameters of this new part and
does the same for all the reference parts. In the end, it inserts an instance of *PartAssembly* with its *features* attribute pointing to all the features of the two data blocks and its *components* attribute pointing to all the new parts and reference parts.

4.4.3. Translator for Inserting Data into CA

The algorithm for this translator is based on the format of the design file. It consists of three major functions—*InsertComponent, InsertAssembly* and *FillFeatures*. They are discussed in this section.

4.4.3.1. Function *InsertComponent*

*InsertComponent* inserts all the feature parameters of a component part into a design file with the same name as the component part. Figure 4-8 presents its algorithm.

```plaintext
Main()
|
| OID = the last instance ID of the DataFile
| IF OID is a PartAssembly
|     InsertAssembly(OID)
| ESLE // OID is a PartComponent
|     InsertComponent(OID)
| ENDF
|
InsertComponent(OID)
|
| name = OID.name
| OPEN a design file with the same name
| print("LoadProcessModel elano9\nNewNote "|name|"\nSetNoteType Component\n");
| FillFeatures(OID);
| print("SaveNote /usr/people/elano/cmedata/"|OID.name|);
| RETURN name;
|
Figure 4-8 Algorithm of Function *InsertComponent*
The parameter OID of InsertComponent is the object id of the component part in the Part21 file generated by the extracting translator.

The Main function selects the last part OID from the Part21 file and checks if it is a component or assembly part to decide which function to call. Function InsertComponent first gets the name of the component part and opens a design file with the same name. Then, it writes the LoadProcessModel sentence into the design file (see section 4.3.2. for details of this sentence). Function FillFeatures is called to insert all the features of this part into the design file. In the end, function InsertComponent writes the SaveNote sentence into the design file.

4.4.3.2. Function InsertAssembly

Function InsertAssembly is used to generate a design file containing all the features of an assembly part and all its components according to a Part21 file. Figure 4-9 presents its algorithm. It has the same parameters as function InsertComponent.

```
InsertAssembly(OID)
{
    name = OID.name
    OPEN a design file with the same name
    print("LoadProcessModel elano9\nNewNote "|name|"\nSetNoteType Assembly\n");
    FillFeatures(OID);
    FOR each c in OID.components
        IF c.element is a PartAssembly
            partName = InsertAssembly(c.element)
        ESLE
            // c.element is a PartComponent
            partName = InsertComponent(c.element)
        ENDIF
        print("AddCostNoteRef "|partName|" Parent "|name|"\n");
    END FOR
    print("SaveNote /usr/people/elano/cmedata/"|name|);
    RETURN name;
}
```

Figure 4-9 Algorithm of Function InsertAssembly
First it opens a design file with the same name as the assembly part. Then, it inserts the `LoadProcessModel` sentence and all the features of this part. The `FOR` loop is used to insert all the components of this assembly part by calling function `InsertAssembly` or `InsertComponent` recursively. In the end, it inserts the `SaveNote` sentence.

### 4.5. An Example of CA Merge

The flow chart for applying the two translators and the internal merge to CA is shown in Figure 4-10. The number in parentheses of each circle references the section.

![Flow Chart the Two Translators and Internal Merge](image)

**Figure 4-10 Flow Chart the Two Translators and Internal Merge**

In this figure, the extracting translator gets the information of a CA part from a report file and puts it into a Part21 format file (*Part21 File1*) based on the EXPRESS schema in Figure 4-4. This Part21 file will be used as the original target data by the internal merge algorithms. The *Part21 File2* is the consistent view of information that is generated by the external merge and will be used as the source. The *Part21 File3* is the final result of the internal merge, which is the updated target data and will be read by the inserting
translator that puts the data into a design file. Then the CA can read this design file and update its local data automatically.

The data file of the merge rules is based on the schema in Figure 4-4. Entity *Part* becomes an *IndependentEntityRule* instance in the merge rules data file. Its *name* is a key attribute and its *features* is a pointer information attribute which points to a set of *Feature AttribEntityRule* instances. Entity *PartAssembly* and *PartElement* are *ChildEntityRules*, since they inherit from *Part*. Entity *PartNode* is a *DependentEntityRule*. It has two dependencies: one pointing from *PartAssembly* and the other pointing to *Part*. Entity *Feature* and *Parameter* are *AttribEntityRules* and entity *IntegerParameter*, *RealParameter* and *StringParameter* are *ChildEntityRules* inheriting from *Parameter*. For entity *Feature*, *FType* is a key attribute and *parameters* is a pointer information attribute, which points to a set of *Parameter AttribEntityRule* instances. For entity *Parameter*, *name* is a key attribute and *value* is an information attribute of its three children.

Figure 4-11 presents an internal merge example. The Part21 file of the source data is on the top. The Part21 file of the original target data is at the bottom.

In this example, the source represents an assembly part ('JunkAssembly') that has one feature ('Process') having one parameter ('LaborRate'). It consists of one 'Flange', which is a component part that has one feature ('Process') having two parameters ('CommercialLaborRate' and 'ToolingLaborRate'), and two 'Bolt', which is also a component part that has one feature ('Process') having one parameter ('MilitaryLaborRate'). The target represents the same assembly part ('JunkAssembly'),
but it just has one component part ‘Bolt’ and the value of its parameter ‘LaborRate’ is 56.000 instead of 56.500 in the source.

```
#500=TextParameter('LaborRate','56.500');
#510=Feature('Process',(#500));

#520=TextParameter('MilitaryLaborRate','66.000');
#530=Feature('Process',(#520));
#540=PartElement('Bolt',(#530));
#550=PartNode(#540,2);

#560=TextParameter('CommercialLaborRate','56.000');
#570=TextParameter('ToolingLaborRate','65.000');
#580=Feature('Process',(#560,#570));
#590=PartElement('Flange',(#580));
#600=PartNode(#590,1);

#610=PartAssembly('JunkAssembly',(#510),(#600,#550));
```

```
#10=TextParameter('LaborRate','56.000');
#20=Feature('Process',(#10));

#30=TextParameter('MilitaryLaborRate','66.000');
#40=Feature('Process',(#30));
#50=PartElement('Bolt',(#40));
#60=PartNode(#50,1);

#70=PartAssembly('JunkAssembly',(#20),(#60));
```

Figure 4-11 Part21 Data Files of CA Example

The internal merge algorithm starts by merging all the IndependentEntityRule (Part) instances and their ChildEntityRule (PartAssembly and PartElement) instances. So instance #610 (‘JunkAssembly’), #540 (‘Bolt’), and #590 (‘Flange’) are selected from the source and merged with the target. For instance #610 from the source, function Find is called and its corresponding target instance #70 is returned since they have the same key attribute value – ‘JunkAssembly’. Then in function MergeEntityInstance, function MergeInfoAttribs is called to merge the information attribute of these two instances, which is features, in this case. When MergeInfoAttribs finds out that this attribute is really an aggregate of PointerRules, MergeAggrePointer is called to merge its
AttribEntityRule instances and this function calls MergeEntityInstance to merge the Feature entity instance #510 from the source with #20 from the target since they have the same key attribute value - ‘Process’. Then MergeInfoAttrs is called again to merge attribute parameters in entity Feature and MergeEntityInstance is called again to merge entity TextParameter instance #500 from the source with #10 from the target. The value of the simple attribute value is copied from the #500 in the source to the target. After this step, the target appears as in Figure 4-12. Compared to the original target, the only change is the value in #10, which was updated to 56.500.

```
#10=TextParameter('LaborRate','56.500');
#20=Feature('Process',(#10));
#30=TextParameter('MilitaryLaborRate','66.000');
#40=Feature('Process',(#30));
#50=PartElement('Bolt',(#40));
#60=PartNode(#50,2);
#70=PartAssembly('JunkAssembly',(#20),(#60));
```

**Figure 4-12 the Target Data File**

For instance #540 (‘Bolt’) from the source, Find is called and #50 is returned from the original target. After a similar merging process, the target stays the same as in Figure 4-12.

For instance #590 (‘Flange’) from the source, Find fails and CreateEntity is called to create one in target. CreateEntity first creates a PartElement entity instance with the same name ‘Flange’. Then for attribute features of this instance, function CreateAggreEntity is called to create all the AttribEntityRule instances pointed to by this attribute by calling CreateEntity recursively for each of them. So all the Feature entity instances that are pointed to by ‘Flange’ and all the TextParameter entity instances that
are pointed to by the Feature instances are created. After this step, the target is as shown in Figure 4-13.

```plaintext
#10=TextParameter('LaborRate','56.500');
#20=Feature('Process',(#10));
#30=TextParameter('MilitaryLaborRate','66.000');
#40=Feature('Process',(#30));
#50=PartElement('Bolt',(#40));
#60=PartNode(#50,1);
#70=PartAssembly('JunkAssembly',(#20),(#60));
#80=PartElement('Flange',(#90));
#90=Feature('Process',(#100,#110));
#100=TextParameter('CommercialLaborRate','56.000');
#110=TextParameter('CommercialToolingLaborRate','65.000');
```

**Figure 4-13 the Target Data File**

After merging all independent entity instances and their children, internal merge picks the dependent entity instances to merge. So PartNode entity instances #550 and #600 are selected from the source. For #550, Find returns the corresponding entity instance #60 from the target since they point to the same PartElement ('Bolt') and the same PartAssembly ('JunkAssembly') points to them. Then MergeInfoAttribs copies the value of attribute numberRequired from the source to the target, which is 2 in this case. For #600 from the source, a new identical instance is created in the target and its dependencies are also created. After this, the target is shown in Figure 4-14 and internal merge is terminated.

Note that in this example, the values of the instanceMergeRule, inheritanceMergeRule and aggregateMergeRule in the merge rules data file will not affect the final result, since each instance in the original target has a corresponding source instance and there is no aggregate attribute value in these data files.
Figure 4-14 the Final Target Data File
Chapter 5
Conclusions and Future Work

5.1. Review of Objectives and Accomplishments

This thesis presents the architecture, algorithms and implementation to update a local database with the consistent external view of the data, which is termed internal merge and belongs to the synchronization of heterogeneous manufacturing databases.

Three powerful software tools, the EXPRESS schema language, EQL query language and TNR language, make the manipulation of the data from the OODBMS constructed on the ISO10303 possible. The implementation of the internal merge algorithm is accomplished by the full-utilization of the above three languages.

After iteratively designing the data structures of the merge rules and target database information and improving the algorithms of internal merge to make the TNR code more efficient and concise, a successful running program for internal merge has been developed.

The testing strategy after software development, involved construction of several complicated models for a complete investigation of all aspects of the design, such as considering different cases for merging the inheritance entities, updating the relations of a data entity instance, comparing the dependencies of two entity instances, merging the information attributes, etc. After complete testing of the internal merge program and correcting many design flaws, the final version of the program has been found to work for all the cases that the algorithm demands.
One basic question for any DBMS related program is the efficiency of the data search. In the EXPRESS schema, one data entity can point to other data entities. EQL allows the user to utilize OIDs for data search. These features provide a way to get data instantly by using OIDs or tracing the pointers between several entities. Compared to any other RDBMS, the use of OID in database schema design and data search improves the efficiency dramatically without joining several tables together by their redundant foreign keys and going through all the data items in each table to get the wanted data.

In the efficiency testing of the TNR code, a database with 1,000 data entity instances was constructed to be the source and around the same number for the target database. The computer time for internal merge execution was found to be less than 10 seconds. This is definitely much faster than any other program based on RDBMS algorithms.

5.2. Future work

The external merge is another aspect of the synchronization of heterogeneous databases. Its basic task is to construct the neutral database, which contains the common information of all the local databases of independent developed manufacturing software systems. It utilizes all the local databases as the source to update the neutral database, which is the target. Each time when it merges the source with the target in the external merge, it uses algorithms similar to that developed in the internal merge such as algorithms for merging entity instances at different inheritance levels, finding corresponding entity instances, merging pointer attributes and aggregate attributes, etc.
What makes the external merge different from the internal merge is that it merges multiple source data files with the target data file. External merge needs to integrate multiple values of an attribute from multiple sources to determine a consistent value to update the corresponding value in the target. An error should be generated if the source values from the source are out of the allowable range or are inconsistent. A set of merge rules needs to be established to govern the process of external merge. Based on these merge rules, new algorithms for merging multiple values will be developed.
Bibliography


