RIPPLE

A Metadata Repository

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Computer Science
in the
University of Canterbury
by
R. P. Wilson

University of Canterbury
1992
Acknowledgements

I wish to thank Dr. Neville Churcher for supervising my MSc. Thesis. He has provided invaluable aid and assistance throughout the course of this study as well as proof-reading and commenting on numerous discussion documents and draft copies of the thesis.

I also wish to thank my wife, Sandra, for her continual love, support and motivation. Her support has helped me through my thesis trials and tribulations and helped make its completion possible.
Dramatic changes in the way we view software and information systems have occurred during the past 20 years. Manual techniques have been replaced by data dictionary products which are in turn being replaced by computer aided software engineering (CASE) or integrated project support environment (IPSE) systems.

A research and teaching metadata repository system, RJPPL'E, is presented. RJPPL'E represents and manages a flexible and extensible internal conceptual model. This conceptual model is derived by a synthesis of common concepts from a variety of design methods. A layered structure is formed by successive abstractions of the concepts and structures derived by that synthesis. This layered structure provides a powerful metaphor for implementation of both the RJPPL'E repository and design method repository support. Design methods can be defined in terms of this model. Tools to aid the configuration of RJPPL'E to support a wide variety of methods are also presented. Once configured, RJPPL'E can provide repository support to tools implementing these methods. Support for information sharing, tool interaction mediation and other important repository features is also provided.
# Table of Contents

## Chapter One

Introduction.......................................................................................... 1
1.1 A Brief Overview............................................................................. 1
1.2 Motivation....................................................................................... 2
1.3 Objectives....................................................................................... 4
1.4 Introduction to CASE and IPSE.................................................... 6
  1.4.1 A Comparison of CASE & IPSE............................................. 7
  1.4.2 A comparison of RIPPLE with CASE & IPSE..................... 9
1.5 Why does RIPPLE use a Relational DBMS?............................... 10
1.6 Summary ....................................................................................... 11

## Chapter Two

A Survey of Existing CASE and IPSE Systems................................. 14
2.1 Introduction .................................................................................. 14
2.2 Stand-alone CASE tools............................................................... 15
  2.2.1 What Distinguishes Stand-alone CASE Tools?...................... 15
2.3 Integrated CASE Tool Sets.......................................................... 16
  2.3.1 What Distinguishes Integrated CASE Systems?.................... 16
  2.3.2 Information Engineering Workbench (IEW)......................... 17
  2.3.3 Excelerator............................................................................. 18
  2.3.4 Auto-Mate Plus...................................................................... 19
2.4 Integrated Project Support Environments.................................... 20
  2.4.1 Introduction............................................................................ 20
  2.4.2 Ada Project Support Environment (APSE)............................ 20
  2.4.3 Eclipse and Aspect.............................................................. 21
2.5 Current Research Projects............................................................. 21
  2.5.1 SOCRATES .......................................................................... 22
  2.5.2 ESPRIT and PCTE................................................................. 23
2.6 What may we learn from these systems?.................................... 24

## Chapter Three

Design of the RIPPLE Core Model................................................... 25
3.1 Introduction................................................................................... 25
3.2 A Brief Survey of Meta\{1,2,3\} data Requirements...................... 25
  3.2.1 Chen Entity Relationship Method....................................... 26
  3.2.2 Extended Entity Relationship Method................................. 27
  3.2.3 Data Flow Diagramming Method........................................ 27
  3.2.4 Further Methods................................................................. 28
### 3.3 Reduction to the Ripple Core Model

3.3.1 Attributes ................................................................. 28
3.3.2 Domains.................................................................. 29
3.3.3 Objects .................................................................... 30
3.3.4 Connections .............................................................. 30
3.3.5 Concept Properties ...................................................... 30

### 3.4 Polishing the Core Model

3.4.1 The Concept Concept .................................................. 31
3.4.2 The Method Concept ................................................... 32

### 3.5 Core System Concept Interactions

3.5.1 Attribute, Domain Association ......................................... 32
3.5.2 Association of Objects with other Concepts .......... 33
3.5.3 Association of Connections with other Concepts ........... 33

### 3.6 A Closer Look at the Core Model

3.6.1 Domain Construction ................................................... 33
3.6.1.1 Domain Complexity ......................................... 34
3.6.1.2 A Domain Construction ..................................... 34
3.6.2 Object Construction ..................................................... 35
3.6.2.1 Creating Objects Via Aggregation ......................... 35
3.6.2.2 Creating Objects Via Generalisation ....................... 37
3.6.2.3 Creating Objects Via Subset Hierarchy ................... 39
3.6.2.4 Abstraction Links ............................................ 40
3.6.3 Connection Construction ............................................... 41

### 3.7 Semantics

3.7 Summary ............................................................................ 42

### 3.8 Summary


---

### Chapter Four

A Fundamental Metadata Model, Its Use & Control .................. 44

4.1 What is Metadata and why is it useful? ......................... 44

4.2 Data ................................................................................... 45

4.3 Metadata ............................................................................. 47

4.4 Meta-Metadata .................................................................... 49

4.5 Meta-Meta-Metadata ............................................................. 52

4.6 Metadata Hierarchy Representation in the Ripple Core System 54

### Chapter Five

Method Definitions ....................................................................... 56

5.1 Method Definitions – An Overview ........................................ 56

5.2 Construction of a Method Definition .................................... 56
5.2.1 Attributes & Domains .................................................. 57
5.2.2 Concept Mappings ...................................................... 57
5.2.3 Concept Property Objects ....................................... 58
  5.2.3.1 Mapping Ripple and Method Concepts ................. 59
  5.2.3.2 Use of Concept Mappings ............................... 59
  5.2.3.3 Construction of Concept Mappings ..................... 60
  5.2.3.4 Which Approach to use? .................................. 62
  5.2.3.5 Identifying and Describing Concept Mappings to
  Ripple .......................................................... 62
5.2.4 Method Concepts ...................................................... 64
  5.2.4.1 Method Concept Properties ............................ 64
  5.2.4.2 Constraints on Method Concepts ....................... 65
5.3 Use of Method Definitions ............................................ 65
5.4 Sharing Method Definition Information ........................ 65
5.5 An Exercise in Method Definition ................................. 65
  5.5.1 Identifying Chen E-R Model Concepts ................. 66
  5.5.2 Ripple Concepts ............................................... 67
  5.5.3 A Comparison between Chen E-R and Ripple concepts ... 68
5.6 An Actual Method Definition Case Study ......................... 72

Chapter Six
Semantics ........................................................................... 73
6.1 What are Semantics? ..................................................... 73
6.2 Why does Ripple need them? ......................................... 73
6.3 Static & Dynamic Semantics ......................................... 74
6.4 Static Semantics ......................................................... 74
  6.4.1 Method Semantics ............................................... 74
6.5 Ripple Static (Core System) Semantics ......................... 75
  6.5.1 Static Core System Checks ..................................... 75
    6.5.1.1 Common Checks ....................................... 76
    6.5.1.2 Domain Checks ....................................... 76
    6.5.1.3 Attribute Checks ..................................... 77
    6.5.1.4 Object Checks ....................................... 77
    6.5.1.5 Connection Checks .................................. 77
    6.5.1.6 Concept Checks ..................................... 77
  6.5.2 Ripple Dynamic Semantics .................................... 78
    6.5.2.1 Object Checks ....................................... 78
    6.5.2.2 Connection Checks .................................. 78
6.6 Importation Semantics ................................................ 79
Chapter Seven

Information Sharing ................................................................. 80
7.1 Why is Information Sharing Useful? ................................... 80
7.2 RIPPLE Information Sharing Support .................................. 80
7.2.1 Import Permission Structures .................................. 80
7.2.2 The Importing Algorithm ..................................... 82
7.2.3 Import Permissions and Method Libraries ................. 84
7.2.4 Method Definition ............................................... 85
7.2.5 Import Permission Creation Constraints .................. 86

Chapter Eight

RIPPLE Concept Demonstration ............................................... 89
8.1 Introduction ................................................................. 89
8.2 The Demonstration Programme .................................. 89
8.3 Demonstrating the RIPPLE Conceptual Model .............. 90
8.4 Demonstrating the RIPPLE Core, and Method Definition Systems ...... 90
8.5 Demonstrating Information Sharing & Interaction Mediation .......... 91
8.6 Describing the Demonstration Methods to Method Definition .......... 91
8.6.1 Describing of E-R Variant Methods ....................... 92
8.6.2 Description of Data Flow Method ......................... 99
8.7 Sharing Information Between Methods .......................... 101
8.8 The Demonstration Model ........................................... 102
8.9 Creating the Demonstration Model .................................. 104
8.9.1 Creating the E-R Variant Model Views .................... 105
8.9.2 Creating the Data Flow Diagram Model View ............ 108
8.10 Summary ...................................................................... 110

Chapter Nine

RIPPLE Implementation Details ................................................ 112
9.1 Introduction ................................................................. 112
9.2 The Domain System ..................................................... 112
9.3 Objects and Connections ............................................. 112
9.4 RIPPLE/Application Interface ..................................... 112
9.4.1 Interfacing Issues ................................................. 113
9.4.2 A Three Tiered Approach .................................... 113
9.4.3 Semantic Issues .................................................. 115
9.5 An Example of Embedded QUEL ................................. 117

Chapter Ten

Conclusions ............................................................................ 120
Chapter One: Introduction

1.1 A Brief Overview

Dramatic changes in the way we view software and information systems have occurred during the past 20 years. New projects have exhibited increasing size, complexity and the need for multiple system models. Data-oriented methodologies have flourished at the expense of process-oriented ones. A number of methodologies — conveniently, if somewhat loosely, described as “information engineering” — are based on the premise that data items and their relationships remain relatively static while applications and the requirements which lead to their construction are subject to greater and more frequent changes. Therefore we need tools and environments capable of maintaining information about data items and their relationships i.e. metadata.

Most currently used methods were developed in the late '60s and throughout the '70s. In the last decade many more have appeared. However, most betray roots in these earlier methods. Some have been subjected to considerable academic interest and study. Successive researchers have re-worked and extended their capabilities. Innovations such as workstations supporting graphical user interfaces and real-world experience from applying methods have motivated many changes. Often new methods or variants have resulted from such work. This progression of enhanced variants forms evolutionary paths. [Chen76], [Mart81], [Mark83], [Mark84], [Mart85] and [Teor86], illustrate such an evolution with the Entity Relationship Model (henceforth referred to as the E-R Model). A similar evolution is evident in the Data Flow Diagram methods.

Unfortunately, many of these efforts are perceived as academic curiosities or as too difficult to integrate into existing systems. Often their results have been less than satisfactory. This has been due to fundamental problems, shortfalls or inherent difficulty and complexity. Others, however, do show promise such as the Nassi-Schneiderman method [Nass73].

Many methods have, to all intents and purposes, remained unchanged since their creation. But they have seen changes in their implementations. Initially, using methods manually¹ was the only option before computer implementations were built. These systems have also seen dramatic evolution, particularly over the last decade as computer technology rapidly became both cheap and powerful. Manual techniques have been replaced by data dictionary products which are in turn being replaced by

---

¹ 'Manual' use typically means paper and pencil use of a method. However, simple computer applications including drawing packages that store model information but provide no data dictionary capabilities also fall into this category.
computer aided software engineering (CASE) or integrated project support environment (IPSE) systems.

The progression of method implementations stems in part from the increasing size and complexity of projects. Also driving this process is the increasingly large software maintenance problem and an increasing level of software reuse in new systems. Manual implementations of methods are effectively usable for only very small problems. The size of current problems and the need for communications between tools makes using a manual version unpalatable. A similar situation now confronts data dictionary products. They are unable to handle large problems and lack necessary complexity management and information sharing abilities [Somm89].

1.2 Motivation

The methods and systems discussed in section 1.1 represent a major portion of the “Information Engineering” world’s evolution. The tools of the information and software engineering professions have evolved considerably from tedious manual drawings of data models to sophisticated integrated software engineering systems. It is apparent, however, that progress to date has not been sufficient to meet the design requirements of today’s software engineering projects. ‘CASE’ has not lived up to expectations,

![Diagram](image)

Figure 1.1: Tool, Repository and DBMS interaction in an "idealised" repository based system. Transparent information sharing and tool interaction mediation are handled by the repository system.

Great advances have been made in the documentation of data structures while techniques for representing and using metadata, or data describing or characterising data, have been developed. Metadata has been used to aid the description of information from different data models, such as the E-R and Data Flow models, and facilitate information exchange between them. The twin goals of developing quality
software and information systems and the management of their evolution require this data.

One major advantage of systems using metadata is their potential to allow a number of tools, covering all phases of the software development cycle, to use this metadata in a co-operative fashion. Tools using this information are able to enhance the quality of model information they store and manipulate. Metadata may be stored in a common repository (sometimes referred to as a dictionary or encyclopædia) which may act as a separate, possibly intelligent, entity independent of any tools. Figure 1.1 shows the role such a repository plays in an ‘idealised’ system.

Unfortunately, this potential has not been fully realised. Many so-called CASE tools are simply automated diagrammers unable to profit from the exchange of metadata with other tools. In addition, this situation may be counter-productive with respect to the people involved in the process. Many separate and complex tools require a much larger staff training effort and may meet with resistance due to complexities involved. In other cases, proprietary repository structures are a major barrier to the development of ‘plug-in’ tools to address particular needs.

Many of the currently available CASE tools work in isolation, not interacting with other tools. Little can be achieved in this fashion. Typically, facilities for exporting, or importing, information have been limited and, in some cases, nonexistent. It has proven difficult to encourage vendors to provide such facilities. Many are, understandably, unwilling to do so until information interchange standards are agreed upon. One proposed standard is CDIF ([CADR89], [Orns88]), a set of extensions to the EDIF standard [EDIF87]. This is an ASCII file format designed to cater for a number of different methods. Unfortunately, with many self-interested parties involved, finalising a standard may take years. Even then, dissatisfaction with the result could further delay the time when a useful set of tools is capable of reliably sharing design information.

A repository system independent of proprietary structures and capable of addressing metadata interchange between tool systems could prove to be important to the advancement of both industry and education/research/academia. Such advancement is not only measured in terms of the time required to design and build systems, but also their quality assurance. Such assurance may be gained via more complete and intelligent information interchange, model evolution and maintenance control, and the use of appropriate design tools.
1.3 Objectives

The problems outlined in section 1.2, and other related problems, are being addressed in the research community, notably by such collaborative efforts as the ESPRIT programme [Camp87]. The business community is also very active in this area perhaps implying an idea whose time has come. R/PLE was originally envisaged, and has been developed, as a research and teaching vehicle. From its inception the objectives foremost in its design have, to a large extent, been involved in addressing the issues and problems outlined in section 1.2.

As a research and teaching vehicle R/PLE should be reliant on neither software nor hardware not readily available to the wider research community. Ideally, the lowest common denominator should be used.

The most important part of a repository is its fundamental underlying conceptual model. It must be coherent, consistent, complete and flexible enough to provide the facilities and capabilities required by tools using the repository. A repository without such an underlying model may be unable to provide sufficient tool support. Problems of this nature are difficult to fix without major upheaval and may result in the system being ignored.

![Diagram](image)

Figure 1.2: Diagram showing the N(N - 1) interfaces required if every method is to communicate and interact with all other methods.

While it is desirable that the conceptual model exhibit properties mentioned above, it is also desirable it does not become over-complex. The proposed CDIF standard ([CADR89], [Orns88]) contains some seven sections each describing information structures used by different methods. However, such complexity in the conceptual model of a repository seems unnecessary, indeed undesirable. Figure 1.2 shows how this approach may lead to the requirement of a large number of interfaces between supported methods. This also requires an ever-growing number of interfaces to be implemented for each successive method that is added. CDIF's format is essentially the concatenation of every concept in every method it supports. Its design philosophy is to add the details of new methods to those already present. Doing so further
increases its complexity and every user of the format must be made aware of the changes.

Another approach is to utilise an abstraction capable of describing most, if not all, of a method’s information requirements. The abstraction’s base may consist of a minimal number of conceptual data objects. It may provide basic core structures with the ability to indirectly describe concepts and their properties it cannot directly represent. Such an approach allows new additions to the format without major upheavals to its structure as a whole. Only those parts not representable directly by the abstraction need explicit description. Figure 1.3 shows how this approach requires a small number of interfaces to achieve the same effect as that shown in figure 1.2. For each additional method added only one extra interface needs to be constructed.

A repository should be able to support the requirements of a wide variety of current and future tools, each of which implements a particular method. It seems sensible that the make-up of the tool set supported should be decided by the people using them. A tool set used by one team or company may be very different from that used by another. In short, the repository shouldn’t impose a predefined tool set on tool users. This requires that it be able to support a variety of tools. Neither should the set of tools supported be static. If a method not currently supported by the repository is desired then the ability to add it should be present.

Tools such as Data Flow or E-R Model diagrammers are not alone in requiring access to information stored in the repository. Report-writers, for example, also need access to this information to produce their reports. It is through the repository tool interface that information makes its way in, and out. To facilitate this, the repository/tool interface may be chameleon-like. An E-R tool may perceive the interface as that of an Entity Relationship repository. Similarly a Data Flow tool may perceive it as a Data Flow repository and so on. Each tool has access, via the repository interface, to the information stored on its behalf. Report-writers may use
the interface to access information stored by any number of tools. If the interface is to behave in this manner it must be flexible and extensible.

An objective stated above is the ability of a repository to support a variety of software tools. Each of these may manipulate, via and only via the repository interface, information stored within it. Information sharing between software tools allows one tool to modify information shared with others. These interactions between tools need to be mediated by the repository system to ensure integrity maintenance of the stored information. Figure 1.1 shows how a repository might interact with design tools and the underlying database management system.

1.4 Introduction to CASE and IPSE

Computer Aided Software Engineering (CASE) tools have existed in the software engineering world since the late '60s. The term has been applied to a broad spectrum of systems from simple diagrammers to integrated tool sets. Integrated CASE tool sets (also referred to as CASE workbenches or CASE shells) exhibit similar architectures to that shown in figure 1.4.

![Figure 1.4: An architectural model for integrated CASE systems and workbenches. Diagram from [Somm89], section 18.1.](image)

Integrated Project Support Environment\(^2\) (IPSE) systems are recent arrivals in the software engineering world. They are usually designed around an onion-ring like architectural model. Figure 1.5 shows the structure of such a model. The name IPSE implies a general purpose system usable in the design and construction of varied projects. However, this has not always been the case. While general purpose IPSE systems do exist, special purpose ones have also been constructed. The Ada Project Support Environment (APSE) [Somm89], section 19.6, and the proposals outlined in

\(^2\) Sometimes referred to as an Integrated Programming Support Environment.
[Defe85] are examples of specialised IPSEs that have been designed solely to design and produce systems in the ADA language. Other IPSE systems include Aspect [Hite88], MAJIC [Sutc87], Eclipse [Alde85], IPSE [Sell85] and I-Star [Dows87].

Figure 1.5: An 'onion ring' layered architectural model for IPSE systems. Diagram from [Somm89], section 19.0.

1.4.1 A Comparison of CASE & IPSE

Historically, CASE systems have been stand-alone or simple tool sets. They are often designed to perform a single task such as Data Flow Diagramming with data dictionary support. Many stand-alone tools work in isolation with rudimentary information import and export capabilities. Co-operation between such systems is poor and they often have radically different interfaces. In some instances they are able to work in concert with others, however, integration of tools has often been a feature of those supplied by a particular vendor. Other vendors have found it difficult to produce tools capable of integration with such systems.

Examples of stand-alone CASE tools are E-R and Data Flow diagrammers, Nassi-Schneiderman [BLUE] and Action Diagram charters. KnowledgeWare: Information Engineering Workbench (IEW), CADRE: TeamWork and Index Technology: Excelerator provide examples of integrated CASE tool sets.

Currently, CASE systems exist that are capable of generating code from design specifications. Most of these either only produce skeleton code or require modifications to the generated code to produce the final system. Code generated in such a ‘write once’ fashion is typically patched as bugs are fixed and the system is extended during its life-cycle. These CASE systems are not easily capable of reverse
engineering changing systems. The importance of re-engineering should not be under­
estimated ([Zelk78]. Re-designing and re-implementing systems rather than applying
patches provides better quality assurance for the end result. Also, systems resulting
from such an approach are more readily modified to suit changing requirements.

Figure 1.6 shows how better quality assurance may reduce the cost of software
systems. Better quality, especially in the requirements, specification and design
phases, can significantly reduce the cost of later maintenance [Zelk79].

![Figure 1.6: Breakdown of effort required to produce and maintain a system. Diagram from [Zelk78], page 9.](image)

As the development of a system proceeds, the cost to correct any error increases.
In the preliminary design phases errors are very cheap to correct while errors during
operation, such as data loss, may be very expensive. Figure 1.7 shows how cost
increases the later an error is discovered.

IPSE systems provide a common environment within which activities such as Data
Flow diagramming are supported in a consistent and coherent manner. Some
integrated CASE systems now provide similar environments. Often based on a
DBMS, IPSE systems provide an integrated repository in which design information is
stored. The repository is overseen by an object management layer responsible for the
management of all objects held within the repository. Version management,
consistency checking and mediation of tool interactions are all part of this layer’s
function. The main distinguishing feature between CASE and IPSE systems is the
lack of this layer. CASE repositories are essentially a collection of the separate tool
repositories rather than a single integrated one. Overlapping sections within the
repository occur more by chance than by design.

The environment provided by an IPSE is, perhaps, the biggest asset next to the
repository and object management layers. The ability to move from analysis & design
to coding to testing phases etc. all within the same environment is a major advantage.
Transparent information sharing between ‘tools’ implementing such phases places the usefulness and abilities of IPSE systems far beyond those of CASE systems.

![Diagram showing relative cost of fixing errors versus phase of development.](image)

**Figure 1.7:** The relative cost of fixing errors versus phase of development. The upper and lower curves represent a 95 percent confidence interval. Diagram from [Shoo83], pg. 15.

Figures 1.4 and 1.5 show basic architectural differences between integrated CASE tool sets and IPSE systems. Integrated CASE systems typically have a set of tools utilising some common repository but with tools themselves being distinct from one another. The IPSE structure is a layered one where tools work beneath a seamless environment. It may become difficult to distinguish the boundaries of these tools.

### 1.4.2 A comparison of **rippple** with CASE & IPSE

**rippple** is neither a CASE nor an IPSE system. Rather, it is a system that could replace the data dictionary or encyclopedia facilities used by such systems. It does not, itself, provide facilities such as ER modelling capabilities. **rippple** represents and manages a flexible and extensible internal conceptual model. Other methods can be defined in terms of this model. Tools exist to aid the configuration of **rippple** to support a wide variety of methods. Once configured, **rippple** can provide repository support to tools implementing these methods. Support for information sharing and tool interaction mediation are also provided.

It is more than just a data dictionary system though. In terms of figure 1.5 it represents the database, object management and public tool interface sections. In terms of figure 1.4 it represents the central information repository and all necessary
interfacing layers. It is in fact, closer in kinship to the structure depicted in figure 1.1 than those shown in figures 1.4 and 1.5. Here, RIPSLE would exist as the repository system, interfacing with a database system below it, and liaising with tools using it for information management. This diagram ignores the role of an operating system and facilities for ‘trusted’ clients using the repository.

Sommerville’s onion ring model provides a close structural model for RIPSLE but for one main difference shown in figure 1.8. Here, the object management layer has an internal, stratified structure. The database, operating system and tool layers are identical to those shown in figure 1.5. At the lower level of the object management layer the metadata (data about data) used by RIPSLE is stored. Not only does RIPSLE use metadata, but also meta-metadata (data about data-models). This occupies the middle level of the object management layer. The top level, meta-meta-metadata, might be considered to contain data about repository models such as RIPSLE. Some tools interfacing with the object management layer may be capable of using more than one level within it.

In reality this progression of abstractions must eventually come to an end. For RIPSLE the meta-meta-metadata level is this end. Here, the data structures are locked down – they are themselves the meta-meta-metadata. They, in their turn contain the meta-metadata that describes the metadata and so on. The levels of this structure are explained in detail in Chapter Four.

![Figure 1.8: An expansion of Sommerville's onion ring IPSE model (figure 1.5). This diagram shows how RIPSLE introduces structure into the object management layer. Each layer is an abstraction of the layer beneath it. The manner in which these layers are derived is the subject of Chapter Four.](image)

1.5 Why does RIPSLE use a Relational DBMS?

At the inception of this project it was decided to use a relational database as the basis for the repository. RIPSLE has been implemented on top of Ingres ([Ston85], [Date87]) using embedded QUEL ([Ston76], [Ston85]). But why a relational system? Why not implement it on top of Prolog and use its inference engine? The reasons for using a relational database system over others are numerous:
Relational database management systems are already used in a number of existing CASE and IPSE systems such as IEW [Know] and DEC’s CDD/Repository [Naec91].

Ingres is widely-available, off-the-shelf technology. It possesses a well defined query language that may be embedded in a variety of host languages. In this case C has been chosen as the host language.

Relational databases provide low level integrity management through built in locking, concurrency, checkpointing and transaction facilities. Multi-user access, security of stored information and the ability to handle the large volumes of information generated in the development of systems are further advantages of relational databases.

Implementations of both C and Ingres are available on a wide variety of platforms. This provides a high degree of portability. Ease of porting, modification and extension of RIPPLE are important considerations, especially as RIPPLE is an academic exercise.

Tools require access to RIPPLE to manipulate stored information. They should not manipulate information in the repository directly, but should pass instructions to it via an interface. It should behave as a server independent of the tools using it. The Sun Remote Procedure Call (RPC) [Sun88] system provides a simple and convenient way of implementing such an interface (discussed in Chapter Nine).

Version management tools are used within IPSE systems to track changes made to many different types of objects. These include design objects such as Entity Relationship entities, specifications & requirements, source files etc... In light of this a repository must be capable of providing version management services. Relational databases perform these activities much better than, for instance, a Prolog system.

1.6 Summary

In this chapter the background, motivation and objectives of RIPPLE have been discussed. An introduction to CASE and IPSE has been presented and a comparison made with RIPPLE itself.

Chapter Two contains a review of the “state of the art” in the CASE and IPSE fields. Examples are examined, compared and contrasted with other systems and perceptions of their respective ‘ideals’. They are interpreted in relation to the objectives of this thesis to determine what is worth preserving from these systems and what is lacking.
Chapters Three & Four examine how the RIPPLE's conceptual data model was conceived and derive a hierarchical model, dubbed a 'metadata hierarchy'. This is used as the basis of a powerful abstraction mechanism used in conjunction with RIPPLE's conceptual data model.

In Chapter Five the concept of a method definition is proposed. Its use and usefulness are explored with special emphasis placed on the relationship of 'method definitions' with the RIPPLE repository system.

Chapter Six goes on to explain the roles and uses of semantics in RIPPLE while Chapter Seven discusses the information sharing facilities provided by the RIPPLE system.

Chapter Eight demonstrates the method definition system and a small application (named Toolset). The method definition tool is used to create definitions for the Chen E-R model, the Teorey, Yang & Fry Extended E-R model and the Gane & Sarson Data Flow model. Toolset is then used to create a simple model and demonstrate the capabilities of the RIPPLE repository.

<table>
<thead>
<tr>
<th>Method Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen Entity Relationship</td>
<td>Chen Entity Relationship Method</td>
</tr>
<tr>
<td>Extended Entity Relationship</td>
<td>Extended Entity Relationship Method</td>
</tr>
<tr>
<td>Ripple System Administration</td>
<td>Ripple System Administration</td>
</tr>
</tbody>
</table>

Figure 1.9: Title screen of the method definition system.

Figures 1.9 and 1.10 show snaps from Method definition and Toolset systems respectively. They show the initial screens presented to the user. In each of these systems the user selects a method in which to define concepts and the like (in Method Definition) or to manipulate a model (in Toolset).
Chapter Nine provides an in depth look at implementation issues facing the construction of RIPPLE repository system. Further, this chapter describes the extensible interface that allows tool applications to treat RIPPLE as a repository server and make use of its services.

Finally, Chapter Ten concludes the thesis with a discussion of how this project has met and achieved the goals set for it. Further, a brief discussion of potential applications and extensions that may be made to RIPPLE is presented.
Chapter Two: A Survey of Existing CASE and IPSE Systems

2.1 Introduction

A great many systems claim to be CASE tools. A recently completed survey, [Schm90], lists 73 different tools and systems available today. The surveyed systems cover a wide range of hardware platforms and levels of sophistication. Included are well known systems such as IEW by KnowledgeWare and Excelerator by Index Technology. CASE technology has pervaded a large section of the MIS community. The extent and impact of this has been the subject of yet another report, [CSDP89]. Despite the small number of questionnaires returned, this report illustrates some of the problems with current CASE technology and its use in the industry.

This chapter attempts to gain a perspective view on the world of CASE and IPSE systems. It reviews examples from commercially vended systems and the research community. In particular, this exercise will highlight not only current trends and directions, but also neglected or overlooked areas in systems. Such areas are of great interest in the design of a new repository. They provide a catalogue of mistakes and oversights - a useful guide. In addition, it provides the motivation to make sure the conceptual model design for RIPPLE is capable of redressing these mistakes and oversights. Failure to do this begs history to repeat itself.

A number of CASE and IPSE systems are reviewed in this chapter. Information on each has been gathered from a variety of sources. These include publications, surveys, industry pamphlets and advertising material. Advertising material is of dubious use due to its rather biased nature. However, it is sometimes useful to compare claims and statements made in this material with others present in the literature. Unfortunately, space prevents an exhaustive and comparative review of CASE and IPSE systems. In light of this systems which have been included are considered to be representative of those available today.

Section 2.2 considers a selection of stand-alone CASE tools and reviews inadequacies and problems inherent with them. Section 2.3 advances to the more sophisticated integrated CASE tool sets. These systems are considered to be at the forefront of CASE technology today - if only by virtue of being market leaders. Section 2.4 considers current, commercially available, IPSE systems. Finally, in section 2.5, a number of research projects are surveyed, particularly in the IPSE
research area. Such research projects are good indicators of future trends and directions in software engineering industry.

2.2 Stand-alone CASE tools

2.2.1 What Distinguishes Stand-alone CASE Tools?

A stand-alone CASE tool is very much as the name suggests – it stands alone. Such a tool acts in isolation from any other tool – even though it may be working in conjunction with other tools upon the same project. These tools maintain their own private databases or data dictionaries containing design information. In this way some information may be highly redundant between data dictionaries. Redundancy, if not correctly managed, allows inconsistencies to occur with respect to information stored by other tools, stand-alone or not.

The seriousness of problems such as redundancy should not be underestimated. In the Chen E-R model alone a moderately complex problem may contain some hundred entities, five hundred to a thousand attributes and perhaps more than a hundred relationships. This volume of information is not difficult to control and maintain within the confines of a single system. But if another system, such as a Data Flow modeller, is also being used, a significant proportion of this information will be explicitly duplicated between them. Failure to control this redundancy has serious repercussions in later stages of the software’s life cycle.

Stand-alone tools often rely on clumsy techniques to import and export design information where such abilities are present. Predominantly such information is exported from the tool to a file in a particular format. The receiving tool must then read the file to import the information. Because of the fixed file format information may be lost in this process. This may be due to the format not being able to express all the required information. This problem is similar to that experienced with data transfer in the GIS industry.

A lack of agreements for interchange formats further complicates matters ([IEDIF87] and [CADR89] are exceptions, but have not been widely adopted). Considerable time and effort may be invested in entering similar data into different tools. Often Herculean efforts are required to ensure a high level of consistency. The existence of complex systems controlling movement and usage of information in software projects are in part due to such problems. Standards organisations such as ANSI and IEEE have been active preparing standards for software engineering documentation and design such as ANSI standards 933 (software requirements specification) and 929 (software test documentation). However, the resulting paper
blizzard that ensues when such systems are used is difficult to manage and requires further tools to automate their use.

2.3 Integrated CASE Tool Sets

2.3.1 What Distinguishes Integrated CASE Systems?

'Integration' becomes a somewhat fuzzy term when it is applied to CASE systems. Early integrated CASE tool sets were little more than a collection of stand-alone tools utilising an agreed upon import/export file format. Each tool maintained a private data dictionary physically separate from all other dictionaries. Almost without exception a single vendor supplied the entire set of tools in use. More modern examples utilise a common repository, or encyclopaedia, for information storage. Information is integrated with that stored by other tools in a single, coherent data dictionary. Information in the repository may be accessed by any member of the tool set. Such access is usually controlled by restrictions within the tools themselves.

Even so, some of these common repository systems exhibit 'partitioning'. That is, different tools use different partitions within the repository to store information. In this way the data dictionaries are physically in the same place, but are logically separated, or partitioned. These require explicit merging stages to resolve redundancies and consistency conflicts. This is illustrated in the Auto-Mate plus CASE system [Croz89]. It appears integrated CASE tool sets are not all they seem, or claim to be.

These systems have regarded the integration concept as applying to the tools, and user interfaces etc, rather than the information they manage. Integration should also apply to the repository where the information is stored. Having many different data dictionaries existing, separately, within the same repository is not integration. Integration on this rather superficial level inhibits building extensibility into both tool and repository systems. This is a significant disadvantage in their design.

Some systems, such as Knowledgewares' IEW, provide a 'public' tool interface allowing access by other tools produced by other vendors to repository services. Unfortunately these public tool interfaces often allow access only to a subset of the services provided by the repository. This is partly due to the 'trusted' status of tools produced by the same vendor. Trusted tools may utilise services not available in the public tool interface [Somm89] chapter 19.

A common approach to providing an integrated CASE tool set is to create a 'Workbench' such as those provided by IEW and Yourdan's Software Engineering
Workbench. However, Martin [Mart88] points out common deficiencies with CASE workbenches including:

- A lack of standardisation making information interchange between different workbenches difficult or even impossible.
- A lack of facilities which allow a method to be tailored or customised to a particular application or class of applications. For example, it is not usually possible for a user to take a built-in rule and replace it with one of their own. Moreover, the tool or workbench may not support the required analysis techniques.

The first of these is a continuation of the information interchange problems inherent to stand-alone systems. The second point underscores the way in which many CASE tool sets, or workbenches, approach provision of design methods. The supplied methods are very often hard-wired with respect to the fundamental concepts and actions within methods. Some, IEW included, allow different diagramming notations to be used, for example, in drawing ER diagrams. This is hardly the same and represents little more than 'syntactic sugar'.

2.3.2 Information Engineering Workbench (IEW)

IEW [Know] is supplied by KnowledgeWare (USA) and is marketed by Ernst Young Tohmatsu. IEW is a PC based product in use since 1986. Its basic structure is of a central encyclopædia into which a range of tools store design information. IEW uses this encyclopædia to integrate the information gathered at each design stage. From it a complete and consistent picture of the application may be built.

IEW supports a methodology that belongs to the 'information engineering' school [Mart84] and [Mart86], which starts at a higher level of abstraction (strategic planning) than software engineering – [Croz89] page 486.

Structurally, IEW is split into a number of 'workstations'. The Planning Workstation develops strategic needs of an organisation and stores this information in the encyclopædia. The Analyst Workstation uses the Knowledge Co-ordinator and Encyclopædia to create a rigorous logical model. The Design Workstation takes this logical model and produces a physical model which may be used by Gamma, a mainframe-based integrated systems generator.

It is interesting to note that IEW requires two systems, working together, to allow the Analyst Workstation to function. Encyclopædia is the system responsible for managing the encyclopædia itself. If the information stored within this is well structured and non-redundant, as is hoped in a repository, why is the 'Knowledge
Co-ordinator’ required? What function does it perform that is not logically part of the analysis or of the encyclopædia system?

IEW provides automated normalisation of data structures and has good support for view integration and consistency checking. This is an interesting claim as normalisation cannot be done properly without details of functional and multi-valued dependencies [Fagi79], [El-M80] and [Kent83]. IEW infers these dependencies from relationships. Inferring functional and multi-valued dependency information from relationships cannot guarantee Boyce/Codd normal form [Kent83] or higher normal forms. BCNF requires details of dependencies between attributes within relations that cannot be inferred from relationships alone. It provides for sub-types and generalisation hierarchies within its E-R modelling framework. Unfortunately relationships are restricted to binary relationships only [Croz89]. Further, the restriction to binary relationships may cause ternary and higher relationships to be represented in fashions that are awkward, or even incorrect.

In [Croz89], IEW has received the most praise. It certainly provides abilities and services that are much more sophisticated than those in Excelerator (section 2.3.3) and Auto-Mate Plus (section 2.3.4). However, the modelling facilities provided by IEW are very static — the only customisation allowed is of diagramming conventions. Additionally, basic features and concepts of methods have been discarded. In the E-R modeller the restriction to binary relationships only must be seen to be a handicap particularly in relation to n-ary relationships.

2.3.3 Excelerator

Index Technology Corporation’s (InTech) Excelerator [Exce] is a PC-based product. It was one of the first products to enter the ‘analyst workbench’ market in 1984. It provides both data and process modelling support and is not linked to any specific method.

Excelerator supports methodologies that start from a general view of the system and then decompose each process until all its parts can be implemented in program code. — [Croz89] page 486.

Excelerator bases its data modelling facilities on the E-R Model and Bachman data structure diagrams. Chen’s original model and another from the Merise system [Cent78] are the Entity Relationship variants used. However, the differences between them are largely cosmetic. Only binary relationships are supported and no provision for relationship membership class is made. Additionally, no direct support for more advanced features present in the Extended E-R Model is present. No normalisation of the resulting structures is carried out by the system. The only
analysis supported by Excelerator is of record description consistency and possible violations of third normal form.

As with IEW (discussed above in section 2.3.2), Excelerator also provides a very static set of modelling tools. Customisation is non-existent and there is no ability to extend the system to support other methods. An example of other problems with such systems is graphically shown in a figure on p265 [Mart85] showing an E-R diagram produced by Excelerator. Over one hundred entities are arranged in a regular grid with relationships forming complex railway tracks in the gaps between them. The diagram is extremely hard to make sense of and would be of questionable value in the form presented.

2.3.4 Auto-Mate Plus

Auto-Mate Plus, developed for the UK Government, was specifically designed to support the Learmouth and Burchett Management Systems (LBMS) [Lear] and the Structured Systems Analysis and Design Methodology (SSADM) [Long87] methods.

Like Excelerator, it enters the life-cycle at the business system analysis stage and provides good support for the analysis and design stages in the form of data and process modelling with an integrated data dictionary and extensive reporting facilities. [Croz89] page 487.

Auto-Mate Plus allows both ‘top-down’ and ‘bottom-up’ modelling methods. Each of these approaches is followed separately. A ‘composite logical data design’ stage is used to merge the two resulting models. This required merging stage strongly indicates a partitioned, or non-integrated, data dictionary. Each partition acts as a separate entity with no reference to the information contained in other partitions. Thus redundancies and inconsistencies must be resolved in the merging stage. Having to merge the two models presents opportunities for problems to occur if collisions between the two models are not handled correctly.

Extensive normalisation and data analysis tools are incorporated into Auto-Mate Plus, but appear to be of little value as they have almost no ‘intelligence’ and are of limited use.

For example, the normalisation tool consists of a dialogue in which the user is led through the various normal forms for each of his entities and is asked to identify repeating groups and undesirable functional dependencies. [Croz89] page 488.

Similar qualities mar the provision of a ‘relation optimisation’ tool. This approach is little better than performing these tasks manually with pen and paper.
Coupled with obvious flaws in the design and use of its data dictionary facilities, Auto-Mate Plus falls far short of the potential of CASE systems.

2.4 Integrated Project Support Environments

2.4.1 Introduction

Section 1.4.2 presented an initial comparison of CASE and IPSE systems. IPSE systems are relatively new in the field of software engineering. Currently there are few commercial examples of IPSE systems. However, there are a number of these systems in development, both in the commercial and academic worlds. There is a high level of academic interest in these systems as researchers push forward the frontiers of software engineering. Particular examples may be found within the confines of the ESPRIT programme.

2.4.2 Ada Project Support Environment (APSE)

Commissioned by the US Department of Defence as the successor to FORTRAN for all DoD contract work, ADA is used to write a large number of complex software systems. During Ada's development the need for an associated support environment was recognised. The Stoneman proposals [Buxt80] envisage an ADA programming environment that is portable and available on a variety of machines. The achievement of this level of portability prompted a three level system of program support to be suggested. A kernel environment KAPSE, a minimal environment MAPSE and the full ADA environment APSE form an architectural model shown in figure 2.1.

Figure 2.1: The organisation of an APSE. Diagram from [Somm89] Chapter 19.
Sommerville notes that the requirements for an integrated generic support environment overlap to such an extent it is likely that Ada environments will be built by configuring general-purpose environments with Ada specific tools. In the decade since the Stoneman proposals were published, a few further proposals have emerged such as [Defe85] which has been agreed upon. However this is lacking in several important facilities. In 1989 development of their specifications was still underway. This serves to illustrate the time required for decision processes regarding the future course of development of these systems.

2.4.3 Eclipse and Aspect

Eclipse [Alde85], [Cart88] and Aspect [Hitc88], [Hall85] are funded and supported by the UK Alvey Programme.

Aspect is termed an ‘IPSE kit’ (in the manner of PCTE [Groo89]) which is subsequently populated by tools such as compilers, debuggers and management tools. It attempts to support the whole software development process in addition to providing an environment for the easy interworking of tools. Objects may be shared between developers and are stored in a central database. Users of the system develop within private areas and may 'publish' versions of objects to be shared by others. All functions provided by the Aspect system are made available through its Public Tool Interface (PTI).

Eclipse set out with the aim of devising a basic structure with its own coherent philosophy, but which is capable of supporting the mainstream methods, notations and languages. Other features emphasized are a derivation procedure mechanism to define system structure in advance of code components and the re-use of existing components. This system is then populated with tools to provide the final IPSE system used to develop software systems. Eclipse is a closed system in that there is no public interface allowing outside tools to make use of information stored in the repository.

2.5 Current Research Projects

Research into integrated CASE tool sets and IPSE systems is currently a hot topic in the academic community. Commercial interests are also very active in this area fuelled by the high level of competition and opportunities in the information technology market place. Volpe, Welty & Company, a San Francisco based CASE research organisation, estimated the CASE market exceeded $US 2.3 billion in 1990 and expects that to rise to $US 5 billion by 1993. They estimate that Digital
Equipment Corporation's CASE programme alone earned more than 10 percent of its total revenues in 1990.

In the IPSE research area the collaborative ESPRIT project is perhaps the largest to date. Other, smaller projects, like SOCRATES, are also contributing valuable research results.

2.5.1 SOCRATES

SOCRATES [Hofs90] project was established by the Software Engineering Research Centrum (SERC) in Utrecht, the Netherlands in June 1989. SOCRATES focuses on improvement of the automated support of information modelling processes. This is realised by the development of a new architecture of CASE tools that reflects both the tasks and models resulting from the modelling process. Thus the final product of the SOCRATES project will be a prototype of this architecture.

Past CASE tools and methods have held a view of the data or process requirements of the system being modelled. It is claimed in SOCRATES that it is necessary to have knowledge of both sides of the requirements engineering. SOCRATES is designed to provide automated support and not replace the user in the modelling process. Guidance and advice is offered to the user by the system based on an explicit strategy extracted from experienced practitioners. By using such information explicit knowledge about information modelling processes is not required.

This approach will result in workbench implementing an information modelling process and containing heuristic decision making and guidance facilities to aid the user. However, these are all based on the opinions and views of the interviewed 'experienced practitioners'. The quality of this information is highly dependent on the quality of the 'knowledge acquisition' and 'knowledge representation' processes. The knowledge representation process itself is based on a specially designed method, called SOCRAMOD, based on NIAM ([Leun87], [Nijs89]) and a rule specification language. Finally MRL (Method Representation Language) is used to describe the resulting system from which a workbench shell is created. Unfortunately, SOCRATES cannot make good use of other methods added to the system without the 'experienced practitioners' to supply knowledge to the heuristic decision making and guidance facilities. In addition, the complexity involved in describing information modelling processes to SOCRATES results in configurations that are not easy to change.

Organisations involved in software engineering activities, and who already have well established information modelling processes in place, may find the strategy
proposed by SOCRATES to be intrusive. Its adoption may require the organisation to change its information modelling process to one supported by SOCRATES. Alternatively, it may have to go through the knowledge acquisition and representation processes before developing the required SOCRATES workbench shell. This may take a considerable time and carry no guarantees regarding the ability of the new implementation to faithfully, or accurately, reproduce its information modelling process. Integration of systems already in use by the organisation with SOCRATES appear to be difficult, if not impossible. Faced with these prospects, such an organisation may well elect to use an alternate system.

2.5.2 ESPRIT and PCTE

ESPRIT is a ten year research programme (1983-1992) of the European Communities. The programme is controlled by the CEC (Commission of the European Communities). Projects are undertaken by ‘partners’ (industries, universities etc...) in at least two different European countries. One partner acts as a ‘prime contractor’ and is charged with supplying the project manager. Together, the ‘prime contractor’ and other partners form ‘the consortium’ [Groo89].

The ESPRIT programme has undertaken to encourage European interest in information technology. It aims to allow the European information technology industry to compete with the rest of the world and to contribute to international standards. Developments in a number of projects within the ESPRIT programme may be found in [Ande91].

The acronym PCTE is derived from the phrase ‘A basis for a Portable Common Tool Environment’. PCTE was a project carried out within the ESPRIT programme from October 1983 until April 1988.

The main objective of the (PCTE) project was to provide a definition of a support environment for software engineering applications. The focus of the PCTE project is on the interface between an operating system and a class of applications. – [Groo89] page 136.

PCTE is dedicated to the operation and construction of software engineering tool environments. It has played a role in many other ESPRIT projects especially those concerned with software engineering environments. Several have the objective of building IPSEs on top of it (see figure 2.2).

![Layered Architecture](image)

Figure 2.2 A layered architecture illustrating the role played by PCTE.
A similar system to PCTE has been developed in the United States of America. Common APSE Interface Set (CAIS), is an interface specification at operating-system level for ADA software development environments [Defe85]. It is designed to allow APSE systems to be more portable between different operating systems.

2.6 What may we learn from these systems?

The tools and systems outlined in this chapter have revealed aspects of themselves that may be learned from. Both strengths and weaknesses have become evident. Repository support, integration at the information and tool levels, conceptual design, extensibility and flexibility of the underlying architecture are key aspects.

The importance of extensibility, particularly within the repository base and its interface, must be stressed. This is lacking in all commercially vended CASE toolsets and workbenches reviewed. Chapter Nine discusses how RIPPLE provides for this repository feature.

Many outlined systems, including proposed standards for information interchange, exhibit considerable complexity in their design. Little attention appears to have been paid to the provision of coherent conceptual models, particularly within commercially vended systems. Reviewed research projects have fared better to varying degrees. This indicates the extra attention paid to design of these systems. Those which do have conceptual models are often complex and formal – type lattices etc compared with the clean and simple conceptual model used by RIPPLE (described in Chapter Three).
Chapter Three: Design of the Ripple Core Model

3.1 Introduction

Before Ripple can begin to look like a repository it must possess an underlying conceptual model. This model describes not only the internal structures used to store both design and repository information but the ways these structures are utilized. As mentioned in section 1.3, constructs with sufficient expressive power and flexibility to cater for metadata requirements of a wide range of methods are required. One set of such constructs is explored in Chapter Four. The structures presented in that chapter demonstrate both an elegant layered form and the feasibility and practicality of such a conceptual model.

We wish to find an abstract model to which other methods may be fitted. How may we produce specifications for this model? One approach could be to take an existing method and base the core model of Ripple upon the conceptual structure of that method. This approach is fraught with questions and problems. Which method should be selected? What criteria should be used to select it? How may we assess its ability to adapt to changing requirements and new methods? Does there even exist such a method? We do not only want the ability to provide the required structural report but also a clean and elegant solution.

If this approach were undertaken, it is likely that an E-R Model variant would be a candidate, if not the choice. Its expressive power, especially in later variants, would certainly cater for many requirements of the conceptual model. However, the E-R model is by no means perfect and questions would remain as to its ability to cope, particularly for process oriented methods. Why pick a particular method as the basis of this conceptual model? Why not have the best of all worlds?

This chapter examines the problem of conceptual model design for the core structures of the Ripple repository. A number of existing methods are examined to investigate the suitability of concepts present in each. From this survey of concepts and structures, the core concepts and structures for the Ripple repository are distilled.

3.2 A Brief Survey of Metadata Requirements

Every method has a specific set of model storage requirements intimately related with its conceptual structure. Entity Relationship methods, for example, need to store information about attributes (properties), descriptions of their domains (value sets), entities (structured groups of attributes) and relationships between entities. Data Flow methods need to store information about processes (that transform data
flows), data flows (flowing between processes), descriptions of the flowing data (data definitions) as well as file stores, sources and sinks etc...

In the remainder of this section, requirements for a number of commonly used methods are examined and dissected. The methods examined are not an exhaustive collection. They were chosen to represent both process and data oriented methods and are well known and understood. Process decomposition diagrams, state transition diagrams, Nassi-Schneiderman diagrams, semantic nets and structure charts are but a few of the vast number of methods which could have also been examined with similar results.

3.2.1 Chen Entity Relationship Method

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value set (domain)</td>
<td>Describes the set of values an attribute may legally hold. For instance, valid ages for people. Value sets are identical to domains [Date86b].</td>
</tr>
<tr>
<td>Property (attribute)</td>
<td>A property (of an entity) capable of holding an atomic value. Legal values are governed by a value set associated with the property.</td>
</tr>
<tr>
<td>Entity</td>
<td>A distinguishable object, for instance, an employee. Entities¹ consist of a number of properties. This method allows entities to be constructed by abstracting a set of properties using the aggregation abstraction mechanism.</td>
</tr>
<tr>
<td>Relationship</td>
<td>An association between one or more entities, for instance, “A company employs many employees”. It may have one-to-one (1:1), one-to-many (1:M) or many-to-many (N:M) cardinalities. These cardinalities (as a whole) are properties of the relationship. Individually, i.e. a one or many cardinality may be thought of as a property of a particular end of a relationship.</td>
</tr>
</tbody>
</table>

¹ An entity is used here to denote an entity set/type rather than an entity instance.
3.2.2 Extended Entity Relationship Method

The Extended E-R Model used here, and referenced in other parts of the thesis, is the variant described by Teorey, Yang & Fry in [Teor86].

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Value Set (domain)</td>
<td>Identical to attributes in the Chen E-R Model.</td>
</tr>
<tr>
<td>Property (attribute)</td>
<td>Identical to properties in the Chen E-R Model.</td>
</tr>
<tr>
<td>Entity</td>
<td>Identical to entities in the Chen E-R Model except that entities may be involved in a number of different abstraction mechanisms. These are aggregation, generalisations and subset hierarchies. Note that aggregation is the only abstraction principle present in the Chen E-R Model.</td>
</tr>
<tr>
<td>Relationship</td>
<td>Identical to relationships in the Chen E-R Model except that entities involved in a relationship may also have a membership class. A membership class may be optional (the entity may, or may not, be involved in the relationship) or mandatory (the entity must take part in the relationship). Membership classes are properties of the relationship. They may, however, also be thought of as properties of the ends of the relationship in the same manner as cardinalities.</td>
</tr>
</tbody>
</table>

3.2.3 Data Flow Diagramming Method

The Data Flow Method outlined here is that described by Gane & Sarson [Gane79].

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process</td>
<td>A process, procedure or action that transforms flows of data, for instance, “Calculate employees salary check”. Data flows both enter and leave processes.</td>
</tr>
<tr>
<td>Data Flow</td>
<td>A directed flow between processes, file stores or external entities (sources and sinks). For instance an employee record flowing from a file to the process</td>
</tr>
</tbody>
</table>
calculating that employees salary check. The flow is annotated with a list of the items moving along it.

**File Store**
A store of data, typically a file stored on some media. The structure of the file is known and described in a data dictionary.

**Source/Sink**
A source, or destination, of data. For instance, an employee providing details for personnel records or accepting salary payment details. Also referred to as an external entity.

### 3.2.4 Further Methods

We could continue to describe a number of other methods. As mentioned above, structure charts and bubble diagrams could have been examined to produce similar results. The reader will appreciate the similarities between the methods examined in preceding sections and these methods in that they all have symbolic representations for ideas of object types and connectivity. The chief differences may be in the way concepts such as attributes and value pools are represented.

### 3.3 Reduction to the Ripple Core Model

Within the methods examined above, there exist many similarities. Classes of concepts from these methods exhibit remarkably similar basic qualities. The idea of a 'property' (attributes or equivalents) with an attendant value set (domains or equivalents), and those containing structures of properties (entities or equivalents) are very common indeed. So too is the idea of connecting ‘things’ together (E-R relationships, Data Flow Diagram data flows or equivalents). Such connections need not always represent a physical, or actual, connection. Forms of logical connectivity between items also fall into this class.

The remaining sections introduce fundamental concepts in the Ripple core model. The reasoning and logic behind each choice is explained with examples of how each relates to their equivalents in the examined methods.

### 3.3.1 Attributes

All the examined methods have the idea, or an equivalent concept, of an ‘attribute’ or a ‘property’. These are capable of storing an ‘atomic’ piece of information and are not to be confused with the idea of a ‘composite attribute’. Examples of such attributes are: a person’s age; 52, their birth date; 19/08/1959, a persons phone number; 298-6664 or an employees status within a company; Manager.
An attribute is usually named and has an appropriate description. Additionally they have either explicitly, or implicitly, a description, or reference to such a description, of the attribute's allowable values (value set).

All E-R variants examined contain, explicitly, the idea of attributes. Other models, such as Data Flow models, tend to contain attributes implicitly. In the case of Data Flow models attributes are apparent within the data dictionary describing data moving along data flows. A process may also contain 'attributes' describing its ID, level or priority — however these are actually properties of the process and not attributes in the sense used here. In the diagraming notation itself no distinction is made between an attribute and a composite structure of attributes flowing along a data flow. Though one is made in the repository, this is completely transparent to the tool implementing the method.

3.3.2 Domains

All of the examined methods having the attribute concept discussed in section 3.3.1 also have the idea, or equivalent concept, of a ‘domain’ or ‘value-set’. This describes the set of values an attribute may draw its value from [Date86b]. Examples of such domains are: Peoples Names; “Character strings 50 characters in length”, Phone Numbers; “Character strings 15 characters in length”, or Employee Types; enumeration of (Managers, Office Workers, Shift Workers) or the age of a person; integer values from 0 to 120. Some domains, such as names, can be very complex. Tasker, [Task88], discusses how to build domains for attributes such as names and how to support them.

Few models appear to, explicitly, contain the idea of a domain. Yet all contain, in one way or another, domains. Merely stating that an attribute contains integer values is imposing a domain upon that attribute. Domains seem to frequent the data dictionaries of design methods and tend to remain hidden in graphical representations.

Domains have been refined into three parts. The idea of a domain still exists but there are two extra parts to it in RFPCE. These are called domain parts. The two varieties are simple and enumerated domain parts. These, in conjunction with domains proper, provide a flexible and powerful facility for describing domains. This is further discussed in section 3.6.1 covering the construction of domains.
3.3.3 Objects

Design methods seldom deal solely with attributes. More often than not they manipulate structured objects such as composite attributes, entities, processes, data flow data to name but a few. The term object is not restricted to the usual Object Oriented sense, even though such objects fall into this category.

In fact objects derived from structuring, or abstraction, are in exceedingly common use within design methods. Differences are chiefly ones of construction. Aggregation is the most common among a number of abstraction principles. Some objects do not possess an internal structure. Some, e.g. Data Flow file stores, do possess other properties. They are distinguished from attributes through the absence of a domain and are considered to be objects because of this.

3.3.4 Connections

The idea of 'connecting' one of more items together is also very common. Examples include Entity Relationship relationships between entities, Data Flow Diagram data flows between processes and Process Decomposition Diagram links between processes.

These examples are physical, or actual, connections. Other, logical, connectivity may also be described by such connections. For instance, some E-R variants contain the idea of mutually exclusive or contingent relationships [Mart81] and [Mart85]. In this way one relationship may hold if, and only if, another relationship does, or does not, hold. Here there exists a logical connection between the two relationships, whereas relationships between entities are examples of physical or actual connections.

3.3.5 Concept Properties

The concepts derived in sections 3.3.1, 3.3.2, 3.3.3 and 3.3.4 above seek to provide the fundamental concepts present in these methods. An attempt to match a fixed set of concepts to concepts present in a range of methods is unlikely to be successful. To this end extra property information needs to be 'attached' to a concept to make it conform more precisely to the requirements of a particular method. A detailed account of how RIPPLE addresses this problem is represented in section 5.2.3.
3.4 Polishing the Core Model

In section 3.3 four basic concepts have been ‘distilled’ from those present in a variety of methods – attributes, domains, objects and connections. These concepts form the basis of the core conceptual model for the RIPPLE repository. A similar approach has been taken in [Zahr81]. This paper describes the IGT model which uses three classes of elements (type, property and connection) to describe the structure of data oriented design models. It uses the idea of a ‘universal data model holder’ for each design method.

From this point on, the convention of using italics to distinguish a RIPPLE core model concept from other meanings of the terms will be used. Thus concept refers to the RIPPLE concept of a concept and attribute refers to the RIPPLE concept of an attribute.

One aspect of the examined methods has not been redressed within the chosen concepts, or their structures. No explicit provision has been made to cater for the varying diagramming notations used by different methods. This is quite intentional. It is not the place of the repository to explicitly provide such facilities. It is, however, the place of the repository to provide, within its framework, the abilities to allow methods to construct such facilities for themselves. This is the path RIPPLE has followed. For a more detailed account of how this may be achieved see section 5.2.3.

The four concepts outlined above do not represent the entirety of RIPPLE’s conceptual model. The remainder are derived from structures present in Chapter Four on a metadata hierarchy for the RIPPLE conceptual model. Two extra explicit concepts take part in the RIPPLE core model. These are the concepts of methods and concepts.

As will be demonstrated later on, these concepts form a powerful base from which to build the structures required by tools using the repository.

3.4.1 The Concept Concept.

A concept in the core model refers to instances of actual concepts within a method. E-R attributes and Data Flow processes are two examples of concepts. The first is an instance of an attribute concept while the second in an instance of an object concept. Later, when a model is being populated with information by a tool, instances of these concepts are created. In this fashion, once the attribute concept has been created in an E-R model then all attributes created within that repository database are instances of that concept.
The idea of a concept is used heavily when describing a method to \textit{RIPPLE}. This process, described in detail in Chapter Five, creates all concepts and their structures present in the described method. Each of these concepts is instantiated from the system concept. In essence, the entirety of a method definition is derived from this concept.

3.4.2 The \textit{Method} Concept.

A \textit{method} in the core model refers to an actual instance of a method storing information in the repository. In some ways this is a concept of convenience and is really only included for book-keeping purposes. Sometimes it is useful to think of concepts in a method being properties of that method instance. For instance, attributes, entities and relationships may be thought of as properties of the Chen E-R method instance. A method must be described to \textit{RIPPLE} before it may be used. Part of that description is the description of the concepts within that method. Once these are described instances of them may be created within the method and used to populate the model database.

3.5 Core System Concept Interactions.

Within the conceptual model, and the functional core system of \textit{RIPPLE}, each of the basic concepts do not exist in ignorance of the others. In fact, they more often than not make use of other basic concepts in the process of constructing an instantiation of that concept. This linkage between, and use of, other concepts not only benefits the process of their construction, but forms some of the basis for the semantic control supported by \textit{RIPPLE}.

3.5.1 Attribute, Domain Association

Without exception, every attribute existing within a model has an associated domain. An attribute's domain is assigned at the time the attribute is created (and may not be destroyed while there exist any attributes using it). Using an association in this way, rather than just having a domain and its definition as part of an attribute, is quite intentional. Were a domain simply to be a part of an attribute then we lose the freedom to have more than one attribute using the same domain without having to redundantly describe the domain. This is especially useful in situations such as reference relationships in E-R methods.
3.5.2 Association of Objects with other Concepts

Strictly speaking, objects don’t actually associate with any other concept. Rather, they make use of them in their construction. For instance, when creating an object representing a Chen E-R entity, all the attributes taking part in that entity are grouped together to form the object. Part of the description of an object concept is the concepts that may be used to construct it. In the example just alluded to Chen E-R attributes are the only concept that may take part in an entity. For some Extended E-R method, a ‘working unit’ may be created holding the relationship between a worker and a machine. In this case both entities and relationships may be grouped together to form an instance of that object concept. Working units could even be instances of a concept separate from a ‘vanilla’ entity concept in the extended E-R method. Thus we may have a ‘vanilla’ entity concept created with attributes and a separate, more complex, entity concept created with (possible vanilla) entities and connections in the example above.

3.5.3 Association of Connections with other Concepts

In the same manner as object concepts, connection concepts do not actually associate with any other concept but use them in their construction. The distinguishing feature here is that the concepts that may reside at the ends of the connection are specified as part of that connection concepts definition. Thus, a relationship in the Chen E-R method connects instances of the Chen E-R Entity concept. Correspondingly, a relationship in the Extended E-R method mentioned in section 3.5.2 connects both instances of the ‘vanilla’ entity concept and the more complex entity concept.

3.6 A Closer Look at the Core Model

This section reviews the ways in which instances of the concepts mentioned earlier in this chapter are used and constructed. Each concept in turn is examined and is supported by an appropriate example showing the procedure by which it is constructed.

3.6.1 Domain Construction

The way a domain is pieced together is based on a simple hierarchic structure. The basic premise is to start from simple building blocks and progress from there. The term 'simple building blocks' refers to simple and enumerated domain parts. From these domains may be constructed which, in turn, may be used in conjunction with other domains, and domain parts, to construct further more complex domains. The
resulting structure is an hierarchic one. The example provided in section 3.6.1.2 explains in more detail how this structuring is used to describe value sets for domains.

3.6.1.1 Domain Complexity

A domain may be viewed as a value set or value pool [Date86b]. Given this view, how do we say one value is in a particular domain, and that another is not? Domains may be exceedingly complex constructs. They may describe simple ranges of numbers, complex collections of non-contiguous values and other number ranges or enumerations of values. It is also useful to state whether a value, or values, are present in the domain (inclusion), or that they are not present in the domain (exclusion).

ripples provides a domain definition system capable of expressing arbitrarily complex domains. These are based on expressions involving domain parts and other domains coupled with grouping operators. These operators are the set union and set intersection operators. Section 3.6.1.2 shows how these grouping operators are used.

3.6.1.2 A Domain Construction

A domain is constructed from a number of domain parts and/or other domains (called abstract domain parts when participating in the definition of another domain). In defining a domain part, what is actually being specified is a rule defining a set of values. From this point, in order to define a domain it is merely a problem of combining domain parts together. This is equivalent to specifying an expression that combines rules that define domain parts. Domains themselves are simply more complex rules and so may also be involved in domain definitions. The final result is a rule of arbitrary complexity describing the required domain.

Once a domain has been constructed it is a simple matter to test whether a value is within the domain or not. Supplied with the rule describing the domain, and a value, evaluating the rule with respect to the supplied value decides the outcome.

This approach to domain definition is an admittedly practical one that aims to define a domain in simple terms. A more formal theory of types is presented in [Danf88] where the opposite approach has been taken.

Figure 3.6.1 below shows an example of how such an expression is constructed. The construction results in the hierarchic structure used by ripples for domain definition. An example is shown in figure 3.6.1. The expression \( G = (A \cap B) \cup (C \cap D) \) describes the structure and the set of legal values of domain G. This is essentially the result of ‘squashing’ the diagram in figure 3.6.1.
3.6.2 Object Construction

One of the most powerful concepts in data modelling is the ability to draw abstractions over details thus allowing the designer to impose structure over 'chaos' and gain a better understanding of a model.

*RIPPLE* provides the ability to construct abstracted objects that consist of any combination of concepts within both the *RIPPLE* core system, and the concepts present in a method. These objects are created via the aggregation, generalisation and subset hierarchy principles and allow complex abstractions to be created. The particular abstraction principles that may be used to construct particular object based concepts are specified in the method’s definition (discussed in detail in Chapter Five). In the following sections aggregation, generalisation and subset hierarchy are used in the sense of Teorey, Yang and Fry [Teor86].

3.6.2.1 Creating Objects Via Aggregation

Aggregation is the bread and butter of conceptual abstraction. It is the simplest, by far the most commonly used and the best understood abstraction mechanism. Aggregation is the process of taking a number of attributes and/or objects (which may themselves be aggregations of attributes and/or objects...) and grouping them together to form a new object. Some methods also provide the ability to aggregate connection information such as relationships. *RIPPLE* can record aggregations of any set of connections, objects and attributes – even domains if this is desired. In this way aggregation could be used to effectively create bins into which items may
be put. Figure 3.6.2 shows an example of an aggregation used to create a working unit.

![Diagram of Foreman and Working Unit](image.png)

**Figure 3.6.2**: A foreman oversees many working units. Many working units are overseen by a foreman. Diagram derived from [Furt86], page 78.

Let us consider now the method used by **RIPPLE** to create objects via aggregation. For instance, suppose we were to create an object with the same structure as the Pascal record detailing a person shown below in figure 3.6.3:

```pascal
Person_Rec = Record
  Person_ID : Longint;
  Name : Record
    Title : string [Title_Len];
    First : string [Name_Len];
    Last : string [Name_Len];
  End;
  Address : Record
    Street : string [Street_Len];
    Town : string [Town_Len];
    Country : string [Country_Len];
  End;
  Phone_No : Record
    STD : Integer;
    Home : string [Phone_No_Len];
    Business : string [Phone_No_Len];
  End;
End;
```

**Figure 3.6.3**: A Pascal record structure describing the details of a person used to illustrate the comparison with aggregation.

We will assume for the purposes of this document that appropriate domains and attributes have been set up already in the **RIPPLE** system.

The record *Person_Rec* is an aggregation of consisting of four parts: *Person_ID*, *Name*, *Address* and *Phone_No*. Of these, *Name*, *Address* and *Phone_No* are all aggregations themselves.

**RIPPLE** uses a bottom-up paradigm for creating abstracted objects i.e. first the smallest sub-components are created by aggregating attributes, then further levels
of abstraction are progressively constructed in a hierarchical manner until the desired abstraction is obtained. Thus to create Person_Record in ROLLBACK it is necessary to aggregate the appropriate attributes to create the Name, Address and Phone_No objects and then to aggregate these together (along with the Person_ID attribute) to produce the Person_Record object.

Figure 3.6.4 illustrates the structure of the resulting Person_Record object.

3.6.2.2 Creating Objects Via Generalisation

Generalisation is a conceptual abstraction that is also commonly used. It is the process of grouping together a number objects which together exhibit a degree of commonality but which also exhibit differences unique to each. Another object is used to collectively describe them with others describing their differences.

Consider the example used to illustrate how to create objects via aggregation shown in figure 3.6.4. A Person has a number of attributes which are detailed in the Person_Record record. However, a person may also have an occupation and be regarded as an employee of a company. Employees may be split into a number of different types for instance: Worker or Manager. An employee description may be along the lines of the Pascal record shown in figure 3.6.5 (which may be included in the Person_Record record):

```pascal
Emp_Type = (Worker, Manager);
Employee_Rec = Record
    Seniority: Integer;
    Salary: Longint;
    Employee_Type: Emp_Type;
    End;
```

Figure 3.6.5: A Pascal record structure detailing attributes common to employees in a company.
If an employee is a worker then we may record further information such as their availability for shift work, union membership etc. If an employee is a manager then we may record further information such as company car type, perk dollar value details etc. These details are dependent on the type of the employee. It does not make sense to include all of these into the description of every employee – managers do not do shift work, and shift workers rarely get company cars...

Thus workers and managers may be generalised to form the employee entity which may contain further information about employees that does not depend on the type of the employee. The Pascal records shown in figure 3.6.6 detail the two different types of employees.

```
Unions = (Clerical, Cleaners, Engineers, ...);
Worker_Rec = Record
  Shift_Work : Boolean;
  Unions : set of Unions;
  etc...
End;

Cars = (Mercedes, Daimler, Porsche, ...);
Manager_Rec = Record
  Company_Car : Cars;
  Perks_Value : Longint;
  etc...
End;
```

Figure 3.6.6 : A Pascal record structure illustrating the comparison with generalisation.
Figure 3.6.7 illustrates the structure of the resulting Employee_Rec entity and how it may be included in the Person_rec entity.

![Diagram of object hierarchy]

3.6.2.3 Creating Objects Via Subset Hierarchy

A subset hierarchy is similar to a generalisation in structure. A subset hierarchy does not, however, use a ‘deciding’ attribute as does generalisation. There is no actual direct link between the objects that are subsets of the ‘parent’ object. RIDDLE links a set of subset objects to their ‘parent’ object by the use of an abstraction link (described above). Abstraction using a subset hierarchy does not so much result in an abstracted object as associate a number of other items as subsets of some other object. The main distinction is that items being generalised must be mutually exclusive while in a subset hierarchy they do not need to be.

People may be split up into a number of subsets, for example Males and Students. If we wish to describe the fact that Males and Students are actually subsets of people then the subset hierarchy abstraction concept may be used. Students and males may be described by the Pascal record structures shown in figure 3.6.8.

The idea of an abstraction link is explained in section 3.6.2.5.
Male_Rec = Record
    Has_A_Beard : Boolean;
    Pro_Fem : Boolean;
    etc...
end;

Student_Rec = Record
    Num_Courses : Integer;
    Credits : Integer;
    etc...
end;

Figure 3.6.8: Pascal records containing attributes for two overlapping sets of people.

A person who is a student has information about the number of courses taken, the number of credit points gained from passed courses plus further student related information. For a person who is a male we may record whether he sports a beard, is supportive of the feminist movement plus any further male related information shown in figure 3.6.9.

![Diagram](image)

Figure 3.6.9: A subset hierarchy showing how the objects `Student_Rec` and `Male_Rec` are subsets of the object `Person`. The idea of an abstraction link is explained in section 3.6.2.5.

3.6.2.4 Abstraction Links

These are used to associate the objects\(^2\) derived from the use of generalisation and subset hierarchy type abstractions with those that contain the 'decider' or 'target' of the abstraction. The link itself is indicated by a flag in the appropriate attribute indicating the type of abstraction that is being linked to it. Both generalisation and subset hierarchies may use the same object as the 'target' of the abstraction. In the definition of the generalisation or subset hierarchy object a reference field contains the identifier of the attribute or object that is its target.

---

\(^2\) We use generalisation and subset hierarchy in the sense of [Teor86].
In the case of generalisations the object that holds all the objects participating in
the generalisation is linked to the (possibly composite) attribute that acts as the
‘decider’. In the case of subset hierarchy abstractions the object that holds all the
objects participating in the subset hierarchy is linked to the object that represents the
superset of the subsets.

The objects mentioned above appear to be a somewhat artificial construct when
first examined. However, they do have counterparts in the methods which utilise
these abstraction principles. For instance, the Extended E-R method described by
Teorey, Yang & Fry [Teor86] uses the generalisation principle. It uses a box with
pointed ends to indicate the generalisation and another similar construct for subset
hierarchies.

3.6.3 Connection Construction

Sections 3.6.1 and 3.6.2 have discussed the construction of domains and objects in
RIPPLE. A third concept, RIPPLE connections, remains to be discussed. The
relatively simple nature of attributes requires no further discussion at this point.

The derivation of connections is very much as the name suggests. Items
participating in a connection are effectively grouped together. The connectivity
exists between all members in the connection. In a sense, all items in the
connection are connected to each other. The manner and types of items that may
take part in particular connections are controlled by the applicable portion of the
methods’ definition. Method definitions are discussed in detail in Chapter Five.
Figure 3.6.10 shows the comparison between a method’s view of a connection,
and the view RIPPLE has of the same group of connected items.

An EER ternary relationship

Employee  Project

Skill

Entities

A RIPPLE three way connection

Employee  Project

Objects  Connectivity

Skill

Figure 3.6.10: Comparison between a method’s view and RIPPLE view of connections.

"An employee may use many skills on many projects."

In a similar fashion RIPPLE connections may represent other concepts based on
connectivity. Take for example the idea of mutually exclusive, or contingent,
relationships present in some E-R variant methods. Figure 3.6.11 shows such a situation and the comparison between the method and RIPPLE views. The connection implementing the mutually exclusive relationship is a connection in its own right— with the exception that it only connects relationships. This connection itself could be recorded as a property of the relationships it connects (discussed in section 5.2).

![Diagram showing relationships between Person, Female Person Details, Male Person Details, and Person/Female Details Connection](image)

Figure 3.6.11: Comparison between a methods view and RIPPLE's view of a mutually exclusive relationship.

3.7 Semantics

Section 3.6 described some of the ways in which domains, objects and connections may be constructed in RIPPLE. However, the semantics, or the rules, governing these operations have so far largely been left out of the picture.

Semantics play a very important role in design methods. To a large extent they control the mechanics of creating, destroying etc... the information being manipulated. Examples of such semantics are abundant. For instance, suppose the diagram fragment shown in figure 3.7.1 was currently being worked on. We see in this figure two entities—Employer and Employee. Each shares a common attribute—Name. The other attributes shown are to illustrate the entities are indeed different in this example.

What happens if the attribute Name is to be destroyed? Clearly this poses a dilemma. Should this request be denied as the attribute is part of entities in the model? Should it be carried out and the attribute removed also from those entities? If Name is a key attribute in an association entity then should the relationship it represents be destroyed? Rearranged? Split Up? Often semantics are little more than specifying a particular option to take from a list of those available.

It is problems such as these that semantics are used to control. They determine exactly what course of action should be taken for a given contingency. In the example discussed above, the semantics for destroy an attribute might state that the attribute must not be present in any entity in the model. Those semantics might
also specify actions that are analogous to a 'cascading delete'. In such instances, the deletion of some item may require the deletion of some other item or items. In turn, the deletion of those items may require the deletion of some other item or items and so on. In this fashion, semantics controlling what should happen when an item is deleted have caused a 'cascade' of deletions. This will stop when all semantics derived from the deletion of the original item have been satisfied.

Figure 3.7.1 : A simple E-R diagram fragment. "An employer employs many employees".

Semantics as they relate to both methods and the Ripple repository system are discussed in detail in Chapter Six.

3.8 Summary

This chapter has described mechanisms, structures and concepts comprising the conceptual model of the Ripple core system. We have seen that many distinct methods may be viewed in terms of objects, properties, their domains and connectivity between these. Later chapters will show that a particular advantage of this approach is the ability to use this simple, common descriptive form to enable information sharing and extensibility among others.

Now that we have these mechanisms, we can proceed to examine the world of metadata in the next chapter. The way in which data, and the information describing its structure, is examined and related back to the conceptual model is derived here.

It should be noted here that not all abstraction principles are required by all methods and that the construction of objects are unlikely to use all concepts within a method. To this end the particular set of abstraction principles required are specified as part of a method's definition. The subset of concepts that may be used to construct such object concepts are also specified in the method definition for each of these concepts.
Chapter Four: A Fundamental Metadata Model, Its Use & Control

4.1 What is Metadata and why is it useful?

Metadata means, literally, "data beyond or at a higher level from data". Metadata encapsulates the structural information implied in data, but not explicitly recorded within it. It is used by database management systems to aid in the maintenance of database structure and by design tools used to construct data models and software systems. A simple example is "Which attributes exist in which entities?". The answer to this question may be found by examining the structure of the stored data in question, but may not be derived from the data itself.

Just as metadata describes the structure of data, we may also envisage meta-metadata describing the structure of metadata, and further meta-meta-metadata describing the structure of meta-metadata. This forms the basis for successive levels of abstraction. Each level contains metadata describing the structure of information stored one level below (see figure 1.6).

Examples of how a DBMS might use metadata to aid maintenance of database structure and information integrity are abundant – the "Which attributes exist in which entities?" example mentioned above is one such. Consider the many-to-many Supplier-Parts relationship instances of which are recorded in the tables shown in figure 4.1. This is represented by two one-to-many relationships shown in figure 4.2. If the database administrator (DBA) decides to remove a particular supplier or part then the effected shipments (shown in figure 4.3) must also be removed to ensure database consistency. If the data layer only is visible then this requirement is not obvious as no structural information on the database (metadata) is present. However, if metadata for this data is present the DBMS may use it to infer the need to remove affected shipments. It may then advise the DBA of this requirement who may abandon the operation or proceed with a modified course of action.

The layered abstraction from data to metadata to meta-metadata and so on is important to RIPPLE's conceptual model. It relies on information stored at each layer to guide it through operations it performs on information stored by the repository. The example use of metadata mentioned above is a very simple application. The conceptual model of a repository utilises this sort of abstraction extensively.

Throughout this chapter the E-R model is used to describe the structures at each level. Initially a small model fragment is used (a supplier/part relationship).
As each new layer in the abstraction is introduced the structures shown will be those required to support the description of information held in the previous layer.

4.2 Data

The Oxford Reference Dictionary defines data as “Facts or information used as a basis for inference or reckoning or prepared for being processed by a computer”. It is simply information – a persons’ name, a car registration number. It is the raw material stored in a database. Data lies at the bottom of the layered abstraction referred to here as a metadata hierarchy.

The tables shown in figure 4.1 store data about some suppliers and parts. These may only be a very small portion of actual tables. In this case they contain instances of suppliers, parts and which suppliers shipped which parts (shipments). It is worth noting here the naming scheme for items used throughout the E-R diagrams and tables. Items are typically named with a mnemonic noting their origin, followed by their name proper. For instance, E_Name is an entity name; CAD_Name is the name of a concept attribute domain; CRE_Description is the description of a concept relationship end etc...

Some suppliers in the Suppliers relation.

<table>
<thead>
<tr>
<th>S_Number</th>
<th>S_Name</th>
<th>S_Address</th>
<th>S_Phone</th>
</tr>
</thead>
<tbody>
<tr>
<td>49</td>
<td>Smith &amp; Smith Hardware</td>
<td>3252 Tiramea Ave.</td>
<td>3854-228</td>
</tr>
<tr>
<td>72</td>
<td>Hatchell &amp; Co. LTD.</td>
<td>13 Livingston St.</td>
<td>2349-653</td>
</tr>
</tbody>
</table>

Some parts in the Parts relation.

<table>
<thead>
<tr>
<th>P_Number</th>
<th>P_Name</th>
<th>P_Colour</th>
<th>P_Cost</th>
<th>P_Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>17634</td>
<td>Plastic Bolt (10 cm)</td>
<td>Red</td>
<td>$1.78</td>
<td>150</td>
</tr>
<tr>
<td>23175</td>
<td>Gib Board (2.2m x 1.5m)</td>
<td>White</td>
<td>$23.50</td>
<td>75</td>
</tr>
</tbody>
</table>

Which suppliers shipped which parts? – Shipments

<table>
<thead>
<tr>
<th>Shipment#</th>
<th>SH_Number</th>
<th>P_Number</th>
<th>SH_Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>49</td>
<td>17634</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>17634</td>
<td>20</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
<td>23175</td>
<td>10</td>
</tr>
</tbody>
</table>

Figure 4.1: Relations storing supplier, part and shipment details.

The tables in figure 4.1 have a structure that may be described by a simple Entity Relationship model. This model, shown in figures 4.2 & 4.3, contains information that is not representable by the tables in figure 4.1. That is, the relationship between the supplier and part entities, the instances of which are held in Shipments.
The model shown in figure 4.2 contains a many-to-many relationship. This may be broken down by the use of an association entity into two one-to-many relationships. In this example a 'shipment' has been used as the association entity. Note the presence of the SH_Quantity attribute in the Shipment association entity. Figure 4.3 shows the result.

Figure 4.3 describes the tuples shown in figure 4.1 in general i.e. it describes the structure of the entities and their relationships – not specific values representing instances of these entities and their relationships. This diagram may be described by tuples in other tables describing the attributes, entities and relationships present within it.
4.3 Metadata

The Oxford Reference Dictionary defines the prefix meta to be “Prefix denoting a position or condition behind, after, beyond or transcending. From the Greek *meta* meaning with or after”. From this we take metadata in the current context to mean “data beyond or transcending data” or “data describing the structure of data”. Metadata lies one level of abstraction above data in this hierarchy. Tools implementing particular methods store design information at this level.

The tables shown in figure 4.4 store information about a specific Entity Relationship Diagram. In this case the Supplier-Parts diagram shown in figure 4.3.

<table>
<thead>
<tr>
<th>Entities</th>
<th>Attributes</th>
<th>Entity Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>E#</td>
<td>E Name</td>
<td>A#</td>
</tr>
<tr>
<td>1</td>
<td>Supplier</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Part</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Shipment</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>S_Phone</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>P_Name</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>P_Name</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>P_Colour</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>P_Cost</td>
<td>8</td>
</tr>
<tr>
<td>9</td>
<td>P_Qty</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>Shipment#</td>
<td>10</td>
</tr>
<tr>
<td>11</td>
<td>SH_Quantity</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Relationships</th>
<th>Attribute Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>R#</td>
<td>R Name</td>
</tr>
<tr>
<td>1</td>
<td>Supplier/Shipmen</td>
</tr>
<tr>
<td>2</td>
<td>Part/Shipmen</td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
</tr>
</tbody>
</table>
To describe the structure of these tables we may draw another E-R Diagram. This diagram (shown in figure 4.5) contains information that is not representable in the tables themselves.

This diagram describes Chen E-R Diagrams in general, not just specific diagrams. Thus this E-R diagram describes the structure of the Chen Entity Relationship Model itself. Note the inclusion of relationship ends. For relationships as a whole properties such as name and description may be recorded. But separate properties may be recorded for each end of the relationship. An example of such a property is the cardinality of a connection with an entity. We may store the information contained in this diagram as tuples in other tables...
4.4 Meta-Metadata

Meta-metadata describes metadata. It lies two levels of abstraction above data in this hierarchy. At this level, information about the ideas and concepts found in a design method may be stored. These allow us to manage the structure of the metadata in the layer below.

The tables in figure 4.6 (shown below and on the following page) store the description of a specific design method – the Entity Relationship Method.

<table>
<thead>
<tr>
<th>E-R Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>C#</td>
</tr>
<tr>
<td>-----</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>E-R Concept Attribute Domains</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAD#</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>
### E-R Concept Attributes

<table>
<thead>
<tr>
<th>C#</th>
<th>CA#</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>17</td>
</tr>
</tbody>
</table>

### E-R Concept Attribute Descriptions

<table>
<thead>
<tr>
<th>CA#</th>
<th>CA_Name</th>
<th>CA_Dom</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A#</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>A_Name</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>A_Description</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>A_Domain</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>D#</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>D_Name</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>D_Description</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>D_Type</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>E#</td>
<td>1</td>
</tr>
<tr>
<td>10</td>
<td>E_Name</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>E_Description</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>R#</td>
<td>1</td>
</tr>
<tr>
<td>13</td>
<td>R_Name</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>RE#</td>
<td>4</td>
</tr>
<tr>
<td>15</td>
<td>RE_Description</td>
<td>3</td>
</tr>
<tr>
<td>16</td>
<td>RE_Memb_Class</td>
<td>5</td>
</tr>
<tr>
<td>17</td>
<td>RE_Cardinality</td>
<td>6</td>
</tr>
</tbody>
</table>

### E-R Concept Relationship

<table>
<thead>
<tr>
<th>CR#</th>
<th>CR_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Attribute/Domain</td>
</tr>
<tr>
<td>2</td>
<td>Attributes/Entity-Attributes</td>
</tr>
<tr>
<td>3</td>
<td>Entity/Entity-Attributes</td>
</tr>
<tr>
<td>4</td>
<td>Entity/Relationship Ends</td>
</tr>
<tr>
<td>5</td>
<td>Relationship/Relationship Ends</td>
</tr>
</tbody>
</table>

### E-R Concept Relationship End

<table>
<thead>
<tr>
<th>CR#</th>
<th>CR_End#</th>
<th>C#</th>
<th>CRE_Card</th>
<th>CRE_Memb_Class</th>
<th>CRE_Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Many</td>
<td>Optional</td>
<td>Draws from</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>One</td>
<td>Mandatory</td>
<td>Value pool for</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>One</td>
<td>Optional</td>
<td>is a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>Many</td>
<td>Mandatory</td>
<td>is a</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>One</td>
<td>Mandatory</td>
<td>has</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>Many</td>
<td>Mandatory</td>
<td>form</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>One</td>
<td>Optional</td>
<td>is a</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>Many</td>
<td>Mandatory</td>
<td>is a</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>One</td>
<td>Mandatory</td>
<td>form</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
<td>Many</td>
<td>Mandatory</td>
<td>has</td>
</tr>
</tbody>
</table>

Figure 4.6: Tables containing meta-metadata, the structure of which is shown in figure 4.5.

To describe the structure of these tables we may draw yet another Entity Relationship Diagram. This diagram may contain information that is not representable in the tables themselves.
Figure 4.7 shows a structure allowing us to represent a method in terms of its concepts and their properties etc. We may store the information contained in this diagram as tuples in other tables. This diagram describes design methods in general, not just specific methods. It may, for instance, be used to store the description of a Data Flow Diagram or an E-R method. In the case of the Data Flow method we could draw a diagram similar to that shown in figure 4.5. The main differences would be the entities representing the concepts in the Data Flow method and the relationships between them. The tables shown in figure 4.6 would then have details of processes, data flows etc. The reader will appreciate that the structure shown in figure 4.7 is also capable of describing the structure of the Data Flow version of figure 4.5.
4.5 Meta-Meta-Metadata

Meta-meta-metadata describes meta-metadata. It lies three levels of abstraction above data in this hierarchy. This level contains information describing the way in which the ideas and concepts present in methods are stored.

The tables shown in figure 4.8 below store descriptions of design methods in general, and this specific paradigm of design method description (sometimes referred to as meta-modelling).

<table>
<thead>
<tr>
<th>Stored Method Definitions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>M#</strong></td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
</tbody>
</table>

```
Method Concepts

<table>
<thead>
<tr>
<th><strong>M#</strong></th>
<th><strong>MC#</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>

Concepts

<table>
<thead>
<tr>
<th><strong>MC#</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
</tbody>
</table>

Method Concept Properties

<table>
<thead>
<tr>
<th><strong>MC#</strong></th>
<th><strong>MCP#</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>16</td>
</tr>
</tbody>
</table>

Concept Property

<table>
<thead>
<tr>
<th><strong>CP#</strong></th>
<th><strong>CP Name</strong></th>
<th><strong>CPD#</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C#</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>C_Name</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>CD#</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>CD_Name</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>CD_Type</td>
<td>8</td>
</tr>
<tr>
<td>6</td>
<td>CD_Size</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>CA#</td>
<td>1</td>
</tr>
<tr>
<td>9</td>
<td>CA_Name</td>
<td>2</td>
</tr>
<tr>
<td>10</td>
<td>CA_Dom</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>CR#</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>CR_Name</td>
<td>2</td>
</tr>
<tr>
<td>13</td>
<td>CR_End#</td>
<td>6</td>
</tr>
<tr>
<td>14</td>
<td>CRE_Card</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>CRE_Memb_Class</td>
<td>4</td>
</tr>
<tr>
<td>16</td>
<td>CRE_Description</td>
<td>3</td>
</tr>
</tbody>
</table>
```
### Concept Property Domains

<table>
<thead>
<tr>
<th>CPD#</th>
<th>CPD_Name</th>
<th>CPD_Type</th>
<th>CPD_Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CPD_Identifier</td>
<td>Integer</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>CPD_Name</td>
<td>Char</td>
<td>25</td>
</tr>
<tr>
<td>3</td>
<td>CPD_Description</td>
<td>Char</td>
<td>250</td>
</tr>
<tr>
<td>4</td>
<td>CPD_Memb_Class</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>CPD_Cardinality</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>CPD_Numbering</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>CPD_Dom_Size</td>
<td>Integer</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>CPD_Dom_Type</td>
<td>Integer</td>
<td>1</td>
</tr>
</tbody>
</table>

### Concept Relationship

<table>
<thead>
<tr>
<th>CR#</th>
<th>CR_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Method/Method-Concept</td>
</tr>
<tr>
<td>2</td>
<td>Method-Concept/Concept</td>
</tr>
<tr>
<td>3</td>
<td>Concept/Concept Property</td>
</tr>
<tr>
<td>4</td>
<td>Concept Property/Property</td>
</tr>
<tr>
<td>5</td>
<td>Property/Property Domain</td>
</tr>
<tr>
<td>6</td>
<td>Concept/Concept Rel_end</td>
</tr>
<tr>
<td>7</td>
<td>Concept Rel_end/Concept Relationship</td>
</tr>
</tbody>
</table>

### Concept Relationship End

<table>
<thead>
<tr>
<th>CR#</th>
<th>CR_End#</th>
<th>C#</th>
<th>CRE_Card</th>
<th>CRE_Memb_Class</th>
<th>CRE_Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>Many</td>
<td>Optional</td>
<td>Draws from</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>One</td>
<td>Mandatory</td>
<td>Value pool for</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>One</td>
<td>Optional</td>
<td>is a</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>5</td>
<td>Many</td>
<td>Mandatory</td>
<td>is a</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>One</td>
<td>Mandatory</td>
<td>has</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>Many</td>
<td>Mandatory</td>
<td>form</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>3</td>
<td>One</td>
<td>Optional</td>
<td>is a</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>6</td>
<td>Many</td>
<td>Mandatory</td>
<td>is a</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>One</td>
<td>Mandatory</td>
<td>form</td>
</tr>
<tr>
<td>5</td>
<td>2</td>
<td>6</td>
<td>Many</td>
<td>Mandatory</td>
<td>has</td>
</tr>
</tbody>
</table>

### System Concepts

<table>
<thead>
<tr>
<th>MC#</th>
<th>MC_Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Attribute</td>
</tr>
<tr>
<td>2</td>
<td>Domain</td>
</tr>
<tr>
<td>3</td>
<td>Object</td>
</tr>
<tr>
<td>4</td>
<td>Connection</td>
</tr>
<tr>
<td>5</td>
<td>Concept Relationship</td>
</tr>
<tr>
<td>6</td>
<td>Concept Rel_end</td>
</tr>
</tbody>
</table>

Figure 4.8: Tables containing meta-meta-metadata describing the model shown in figure 4.7.
If desired, an Entity Relationship model could be constructed to describe the structure of the tables in figure 4.8. These structures would represent meta-meta-meta-metadada. However, there comes a level at which higher levels of abstraction are superfluous to the task at hand. The last level, presented in this section, is the first abstraction that has become 'too abstract' for the requirements of RIPPLE. It is at this point that the structures used to contain the description of structures at the preceding level of abstraction must be 'nailed down' and the information they contain becomes 'hard-wired' into the system. In the case of RIPPLE, where we wish to store descriptions of methods, the meta-meta-metadada level is where this progression will stop. Thus, the structures described by the tables shown in figure 4.8 will become hard-wired into the RIPPLE repository system.

In many other systems, some of which were reviewed in Chapter Two, the level at which these structures become hard-wired occurs at a lower level than the one in RIPPLE. In other words, these systems stop at the structure of particular methods (i.e. meta-metadada) rather than stopping at the structure of this structure (i.e. meta-meta-metadada).

4.6 Metadata Hierarchy Representation in the RIPPLE Core System

Prior sections in this chapter describe structures required to store information at each layer in the hierarchy. Starting at the data layer and working backwards until the meta-meta-metadada layer is reached, data structures required in the next layer are inferred from requirements present in the current layer. Examining a layer shows structures storing information very similar to, if not the same as, that stored in other layers. The tables containing attribute domains (metadata layer), ER Concept Attribute Domains (meta-meta-metadada layer) and Concept Property Domains (meta-meta-meta-metadada layer) are all isomorphic. These all contain domain type information. In this case there are three sets of structures used to store the same type of information. In fact, these levels contain precisely the objects, properties and connections of the RIPPLE conceptual model. It is easy to see from this comparison that the conceptual model underlying RIPPLE is well suited to describing and storing this very information.

Two possible implementations of these structures exist. The first provides a complete set of relations for each layer. This physically partitions information for each layer from the other layers. However, this requires the management of a much larger number of relations and introduces additional, undesirable, complexity. This extra complexity is unnecessary for the purposes of the RIPPLE core system. The second approach involves collapsing, or merging,
structures storing identically structured isomorphic information in different layers of the hierarchy into single relations. An additional field to distinguish which layer entries belong to completes the process. This utilises one unified set of relations to store information for all required layers of this hierarchy. This is a very natural approach and has been followed in the design and implementation of the RIFFLE core system.

It should be added that both implementations are relatively straightforward and store the same information. The ability to implement the unified approach demonstrates some of the power and flexibility of the RIFFLE core system and its conceptual model. In fact, utilising the view mechanism in Ingres allows effective achievement of both approaches with a single implementation. The RIFFLE core system uses the view mechanism to do just that. Providing the same capabilities with the first approach described is much more difficult and would result in a clumsy design.
Chapter Five: Method Definitions

5.1 Method Definitions – An Overview

*RIPPLE* knows about a very small number of concepts – *attributes, domains, objects* and *connections*. Methods, on the other hand, generally know about many more concepts. Defining a method definition is the process of mapping concepts of a method onto those of the *RIPPLE* conceptual model. Thus the term *Method definition* is taken to mean the description of a particular method, for example, the Chen E-R method.

A method definition encapsulates a method’s concepts, their properties, how to construct them, the relationships between them and the ways in which their instances may be shared. The last part regarding sharing is particularly important. This part of a method definition enables metadata to be shared between tools utilising the *RIPPLE* repository system. *RIPPLE* uses a method definition to create the necessary structures and implement semantic checking to provide repository support for that method.

5.2 Construction of a Method Definition

Before a tool may make use of services provided by *RIPPLE*, the repository system must have a definition for the method implemented by that tool. If an appropriate method definition exists, then the tool may make use of it. The source of such a definition may be one of two places. An existing model database maintained by *RIPPLE* contains within it a set of method definitions. At the very least this set contains all methods that have been used to manipulate metadata stored in the repository system. Additionally it may contain other method definitions that have not yet been used. A second source for an appropriate method definition may be a method definition library. Such a library is nothing more than another *RIPPLE* model database. However, in this case there is no metadata stored within it – only method definitions. Using a method definition from a library involves nothing more than instructing *RIPPLE* to move the definition from the library repository to the current model repository.

If an appropriate method definition does not exist in either the current model repository or a method definition library then the tool is unable to make use of *RIPPLE* until one is provided. The interactive *Method Definition* system is the means by which a new method definition may be created. An example of this process is presented in Chapter Eight.
5.2.1 Attributes & Domains

In sections 3.3.1 and 3.3.2 the concepts of attributes and domains were introduced. In terms of metadata they correspond to instances of concepts such as E-R attributes and E-R domains respectively. At the level the method definition system operates (meta-metadata), the concept of an attribute has a different application and meaning. It is used to define properties of concepts that exist in the meta-metadata layer. As in the metadata layer, domains maintain their close relationship with attributes. In this layer a domain describes the values that may be stored in the attribute representing one of the properties of a concept, or number of concepts, in the method. For instance, we may define a property of an E-R relationship to hold the degree of the relationship. A useful domain to have for this attribute may be integer values greater than one (2 ≤ degree ≤ ∞). Not wishing to leave this open-ended we may restrict it to (degree = 2) for methods that implement binary relationships only.

Once such an attribute has been created, instances of the E-R relationship concept using that attribute may store the degree of the relationship within it. The core system asserts and enforces the property contains only valid values with respect to the domain associated with that attribute. This manner in which the core system does this is discussed in more detail in Chapter Nine

5.2.2 Concept Mappings

Recall that a major feature of RAPPEL is the reduction to the four basic (and hence very general) concepts: attribute, domain, object and connection. These are the basic building blocks a method uses to construct a design with. Concepts present within the conceptual model may be thought of as ‘vanilla’ flavoured. To provide services to methods requiring more complex concepts these ‘vanilla’ constructs are ‘decorated’ with further properties to provide these. Thus concepts may have extra properties assigned to them when described in the method definition system. These extra properties are bundled into a property object (sometimes referred to as a POID – Property Object ID).

This approach is very similar to that used in many Geographic Information Systems (GIS). Such systems possess a small set of spatial object types such as points, lines and polygons (among others). However, GIS systems, by their very nature, must also store attribute information such as a trees height and age in addition to its location. Here the non-spatial attribute tables are ‘hung’ from the small set of spatial object types and act as extra properties of those object types.

The reason for this process is simple. In order to obtain the core conceptual model, the common elements of various methods have been distilled. The
properties decorating the 'vanilla' concepts now represent the distinctive features of
the individual methods. Without these features all methods would look the same
(which would make repository support much simpler!). E-R relationships are a
good example. A name and description (the only properties provided by core
\texttt{Ripple} concepts) for a relationship is not enough information to be useful. At a
minimum cardinality information needs to be stored for relationships as well.

When a concept is created, structures in the core relations are created to store the
information described by the property object of that concept. The binding of a
property object to a \texttt{Ripple} base concept, forming a \texttt{<Ripple base concept,}
method concept\texttt{>} pair creates a concept mapping. Concept mappings represent
interactions \textit{between} the layers shown in the layered object management ring in the
extended version of Sommerville's onion ring model (shown in figure 1.6). In
some methods more than one method concept will be paired with a particular
\texttt{Ripple} base concept. Such many to one mappings require care in the description
and management of property information specified in their property objects.

Consider the case of processes and file stores in Data Flow methods. They both
map onto the \texttt{Ripple object} base concept but each concept has very different set of
properties. Processes need to store their process number and details of sub­
processes within that process. File stores need to store details of the file structure.

5.2.3 Concept Property Objects

As described in the previous section, property objects are used to contain extra
property information for method concepts. Some properties do not, however,
consist of a single attribute. Some lists of items such as the list of key attributes in
an E-R entity. A Data Flow method might store a list of sub-processes within a
process. To allow this type of property information to be stored, property objects
may also contain a specialised object called an \textit{Object Bin}. Such a property
provides the means to store a wide variety of property information that cannot be
held by a finite number of attribute properties. Within an \textit{Object Bin} a tool
application may store any set of items it wishes to such as the examples mentioned
above.

Property objects are created in the same way as any other object within the
method definition system. First, the appropriate domains for the properties are
created. From here, the necessary attributes are created and are associated with the
domains previously created. Once the attributes have been created the property
object is created. Property objects are constructed via the aggregation of attributes,
other property objects and \textit{Object Bins} – other abstraction mechanisms are not
permitted. The attributes, their domains and other property objects may be shared
with other property objects within this or any other method definition. The interactive \textsc{Ripple} method definition and core systems oversee the sharing of these property attributes, domains and objects. If set up correctly, properties may be non-redundantly shared by the methods requiring them.

5.2.3.1 Mapping \textsc{Ripple} and Method Concepts

In the Chen E-R Model (see section 5.5 for mapping details), mappings between method concepts and \textsc{Ripple} base concepts are very straight-forward. E-R attributes map very well to \textsc{Ripple} attributes, E-R entities map very well to \textsc{Ripple} objects and E-R relationships map very well to \textsc{Ripple} connections. In addition to these basic mappings, other information such as relationship end cardinality must be stored as 'property information'.

Mapping Data Flow Diagram concepts to \textsc{Ripple} concepts provides a more complex example. Data Flow attribute information used in the construction of data flow data objects map to \textsc{Ripple} attributes but \textsc{Ripple} connections, this time, map to data flows. However, mapping Data Flow concepts to the \textsc{Ripple} object concept breaks the nice one-to-one mapping of concepts observed so far. There are in fact four candidates for such a mapping:

(a) Processes and their property information
(b) File stores and their property information
(c) External entities and their property information
(d) Data flow definitions and their property information

Note that it makes sense semantically to map all of these method concepts to the \textsc{Ripple} object concept.

How do we manage the situation where a collection of method concepts map to a single base concept in \textsc{Ripple}? Each mapping must be distinguished from all others when manipulating the design. We also need to know which particular mapping we are dealing with when manipulating an item to access the property information for that item. The answer is to produce different concept mappings between the \textsc{Ripple} object concept and design method concepts. Each concept mapping has its own property object.

5.2.3.2 Use of Concept Mappings

Concept mappings provide a simple method for differentiating mappings between design method and \textsc{Ripple} concepts. This allows a tool to access and manipulate the correct property information for a concept instance.

For example, consider file stores in Data Flow methods. The basic \textsc{Ripple} object concept stores the name of the file store and its description. The Data Flow
tool may wish to store other information such as a predicted maximum number of records in the file. The mapping of the RIPPLE object concept to the Data Flow file store concept provides the means to do this. The property object associated with this mapping will contain, among other properties, a property to contain this information. This concept's structure then comprises that of the basic RIPPLE object concept and the property object associated with it.

5.2.3.3 Construction of Concept Mappings

To construct a concept mapping we use the interactive Method Definition system (described in Section 5.5 and demonstrated in Chapter Eight) to associate a property object with a RIPPLE base concept/method concept mapping. This association creates a relation used to store property values for each attribute in the property object. The remainder of this section discusses the implementation of these relations.

Two approaches may be taken to implement these relations with the goal of minimising internal redundancy. These two approaches are distinguished fundamentally by the way in which the property information is grouped.

Consider the following example. A person implementing Method X in RIPPLE decides to create a concept mapping with the following property object:

\[
\begin{align*}
A &: \text{Domain Y} \\
B &: \text{Domain Z} \\
C &: \text{Domain Z}
\end{align*}
\]

Figure 5.1: An example property object containing three attributes based on two domains.

If we create a separate relation for each unique attribute or one for each unique domain we get two different sets of relations:

**By attribute.** Using this approach there will be three relations created. For each unique attribute a relation of the following form will be created:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>i4</td>
<td>This is a generic RIPPLE identifier. It may be an identifier of an attribute, domain, object or connection based concept instance. It is the identifier of the item for which this relation holds values for one of its properties.</td>
</tr>
<tr>
<td>Value</td>
<td>*</td>
<td>This is the actual value of this property. Its value is consistent with the definition of its domain. * The type of this attribute is dependent on the type and size of the domain the attribute draws its values from.</td>
</tr>
</tbody>
</table>

Table 5.1: Structure of property information relations if the 'by attribute' approach is taken.

The criteria for this approach is based on attribute uniqueness. The values for each attribute will be stored in a single relation regardless of how many times and in
which methods it is used. If a method wants to refer to an attribute by a different name then an aliasing mechanism may be used.

**By domain.** Using this approach there will be two relations created. For each unique domain a relation of the following form will be created:

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>i4</td>
<td>This is a generic <strong>RIPPLE</strong> identifier. It may be an identifier of an attribute, domain, object or connection based concept instance. This relation holds the values of all attributes within its property object that share the same domain.</td>
</tr>
<tr>
<td>Attribute ID</td>
<td>i4</td>
<td>The identifier of an attribute within the property object associated with this item’s concept.</td>
</tr>
<tr>
<td>Value</td>
<td>*</td>
<td>This is the actual value of this attribute within the property object. Its value is consistent with the definition of its domain.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>* The type of this attribute is dependent on the type and size of the domain the attribute draws its values from.</td>
</tr>
</tbody>
</table>

Table 5.2: Structure of property information relations if the 'by domain' approach is taken.

As the criteria for creating relations in this approach is based on domain uniqueness then the values for all attributes of a particular domain will be stored in a single relation. This is regardless of how many times and in which methods the domain is used. Attributes are distinguished by the second key attribute in the relation. This approach also allows a name aliasing mechanism to be used if a methods want to refer to the same attribute by different names.

Yet a third approach exists for the design of these relations (but which does not minimise internal redundancy). In this approach a single relation is created to property values. Table 5.3 shows what such a relation would look like.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>i4</td>
<td>This is a generic <strong>RIPPLE</strong> identifier. It may be an identifier of an attribute, domain, object or connection based concept instance. This relation holds the values of all attributes within its property object within a single tuple.</td>
</tr>
<tr>
<td>Value_A</td>
<td>*</td>
<td>This is the actual value of attribute A within the property object.</td>
</tr>
<tr>
<td>Value_B</td>
<td>*</td>
<td>This is the actual value of attribute B within the property object.</td>
</tr>
<tr>
<td>Value_C</td>
<td>*</td>
<td>This is the actual value of attribute C within the property object.</td>
</tr>
</tbody>
</table>

Table 5.3: * The type of this attribute is dependent on the type and size of the domain the attribute draws its values from. Property values are consistent with their domain.
As this approach simply groups all attributes from a property object into a single relation, significant redundancy may result. This redundancy is caused when property attributes are shared between more than one property object. Such a situation occurs when different methods share the same item whose property objects in those methods contain common properties. Obviously, when properties for an item are changed great care must be taken to ensure all other instances of that shared property must also be updated.

5.2.3.4 Which Approach to use?
These alternatives both have advantages and disadvantages. All allow name aliasing mechanisms if required. The first approach gives a larger number of smaller relations (one relation for each unique attribute). The second results in fewer, larger, relations with compound keys (one relation for every unique domain). The third approach creates a single relation for each property object. As mentioned above this may result in internal redundancy of property information. This is highly undesirable and for this reason it is not considered a viable alternative. In spite of this, it is still possible to create relations with the structure shown in table 5.3 using the view operator in Ingres. In this way an imaginary relation combining attributes from number of relation containing properties may be created.

As much of the updating and manipulation of this information is on an attribute basis, rather than a domain basis, the first approach is used in RIFFLE.

5.2.3.5 Identifying and Describing Concept Mappings to RIFFLE
So far this section has discussed the basis and uses of mappings between RIFFLE concepts and method concepts. It has not considered how these mappings are actually identified and subsequently described to RIFFLE. It is important that a method is described accurately and completely to RIFFLE. If there are omissions in the description RIFFLE will not be able to provide comprehensive repository support to that method.

When confronted with a method that is to use RIFFLE as its repository base the person responsible for performing this task must do the following:

1. Identify all the concepts present within the method. In the case of the Chen E-R method there are attributes, entities and relationships that are physically part of a E-R diagram. Further, there is the concept of a domain, from which an attribute draws its values. Domains, while not necessarily taking part in the diagrammatic representation of the model, do exist within its data definition.

2. From the list of concepts derived in (1), split them into two classes:
The first class contains those concepts that represent and/or refer to some fixed item, and only that item, in the method. Entities, attributes and domains from the example above fall into this category. The second class contains those concepts that connect together other concepts. Relationships from the example above fall into this category. So do data flows in Data Flow methods.

The key difference between the two classes is the idea of connectivity. If a concept provides a connective idea between concepts it falls into the second category. If it provides the representation (only) of a concept, then it falls into the first category.

Once this basic distinction has been made within the method's concepts, we may proceed to further categorise them. Concepts within the second class will all map onto the RIPPLE connection concept. From concepts in the first class three separate groups may be simply divided. The groups are characterised in the following fashion:

1. Domain concepts. Any concept whose purpose is to describe a value pool from which attributes (or their equivalents in the method) draw their values is in this group. These map onto the RIPPLE domain concept.

2. Attribute concepts. Any concept whose purpose is to describe the container of an atomic value, such as a part ID or a part's colour is in this group. These concepts are always associated with a domain concept. These map onto the RIPPLE attribute concept.

3. Object concepts. Any concept that stores non-atomic information but does NOT embody connectivity (see above), such as the details of a part but NOT a data flows source and destination processes, is in this group. These concepts may embody structural detail and are often constructed as an abstraction over detail. Other examples are file stores which have no explicit internal structure but which do not have a domain. These map onto the RIPPLE object concept.

Once the concepts present in the method have been identified and grouped as above, the description of the method to RIPPLE may proceed. To aid in this phase, a module of RIPPLE called Method Definition was designed and built. In brief, Method Definition maps concepts in the groups above to their analogous concepts in RIPPLE. It also provides facilities to design and construct property information discussed in sections above. This property information is associated with the method concepts at the time of mapping them to their RIPPLE counterparts. Figure 4.7 shows the relationship between methods, concepts and their properties.

Method Definition is a simple, yet powerful, tool providing the gateway for a method into RIPPLE. In this way rich and diverse method concepts may be
mapped onto the appropriate \textit{RIPPLE} concepts. This provides an accurate and functional description of a method's concepts to the \textit{RIPPLE} repository system. Once this has been provided \textit{RIPPLE} is capable of providing repository services to tools implementing this method.

5.2.4 Method Concepts

The terms \textit{Method concepts} refers to concepts contained within a particular method. They are the ideas and constructs central to its working. Entity relationship methods have domains, attributes, entities and relationships. Data flow methods have data flows, processes, sources and sinks, file stores and so on. These are the concepts the method definition system maps onto corresponding \textit{RIPPLE} concepts.

5.2.4.1 Method Concept Properties

Obviously, \textit{RIPPLE} cannot hope to provide a set of concepts sufficiently rich and diverse enough to accommodate, directly, any method one may wish to describe. In fact, as seen in Chapter Four, quite the opposite approach has been taken; a small and simple, but very flexible, set of concepts have been constructed. With these concepts \textit{RIPPLE} describes method concepts present within a method.

Most, if not all, concepts possess properties. They range from the concept's name and description to permissions lists controlling access. Instances of these concepts possess attributes containing their private values for these properties. These properties are important to the methods which use them in the process of manipulating models.

Some concept properties such as name and description are directly supported by \textit{RIPPLE}, though most are not. However, \textit{RIPPLE} does provide a powerful mechanism for describing these properties and attaching them to concepts. When a concept is created in \textit{RIPPLE} a structure call a property object (described in sections 5.2.3 and 5.2.4) may be attached to it. This object contains all the properties required by the concept. In this fashion any desired set of properties may be created for a concept. An instance of a concept will have attributes corresponding to the properties present in the property object associated with that concept.

Describing a method concept in terms of \textit{RIPPLE} concepts has two main parts. Firstly, the properties of the concept not directly supported by \textit{RIPPLE} must be identified and assembled into a property object. Secondly the \textit{RIPPLE} base concept previously identified as mapping to the method concept being described is selected. Once these two actions have been performed the \textit{RIPPLE} concept representing the method concept may be established. This is done by 'attaching' the property object created in the first step to an instance of the \textit{RIPPLE} base concept decided upon in
the second. The new concept is appropriately named and described via the name and description properties present in the instantiation of the RIPPLE base concept.

5.2.4.2 Constraints on Method Concepts
There are some constraints placed on concepts that make up part of a method definition. These are involved with enforcing semantics within the repository itself and are separate from the constraints placed on these concepts at the time they are created when defining a method definition.

In fact, all these constraints apply to the information sharing capabilities of RIPPLE. In particular, they control which mappings of method concepts between two methods may be involved in sharing information.

5.3 Use of Method Definitions
Method definitions are central to the usefulness of the RIPPLE repository system. Without such a definition for a method RIPPLE is unable to provide repository services to client tool applications. There is another advantage to this approach. A method definition is independent of the RIPPLE repository system. RIPPLE also does not provide any special services of any kind for any method. This means that to describe a method to RIPPLE requires no changes to be made to RIPPLE itself. Further, a method, once defined to RIPPLE, may be moved to another site simply by taking a copy of the method definition to that site.

5.4 Sharing Method Definition Information
Information used to describe a method to Method Definition may be shared with other method definitions. In particular, property attributes of concepts defined in one method may be shared by other methods who also wish to use those property attributes. The advantages and usefulness of this ability are discussed in section 7.2.2.

5.5 An Exercise in Method Definition
In this section the steps required to describe a method to RIPPLE are presented. The method chosen is the Chen Entity Relationship method. This method was selected for its simplicity and the fact it demonstrates every aspect of the method definition system. Its definition involves use of every base concept in RIPPLE and the construction of some simple concept properties.

Initially the Chen E-R and RIPPLE concepts are identified and described. At this point a comparison of concept structures, properties and purposes is made. From this the appropriate mappings are decided upon. These are then described in
detail in a fashion suitable for storage as a method definition in the JPPDE method definition system. A small example based on the ever-popular Supplier/Parts model is used to illustrate the points presented.

5.5.1 Identifying Chen E-R Model Concepts

Chen's E-R Model is the oldest, and simplest by virtue of an absence of extensions, of the raft of E-R variants available. It has four main concepts:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>An atomic value drawn from a value pool. Attributes have a name, domain and description. An example is the identifier of a part. It may be called Part_ID, be drawn upon the domain containing the set of numbers from 1 to 99,999 and be described as “The identifier of a part in the part entity”.</td>
</tr>
<tr>
<td>Domains</td>
<td>A value pool from which attributes draw their values. Domains have a name and a description. This concept is often hidden in the data dictionary describing attributes in the E-R model and has no explicit diagrammatic representation in some E-R variants. Others, such as the variant described in [Furt86] do show domains diagrammatically. The domain used for Part_ID mentioned above may be called “Part Identifiers”, be drawn on the subset of integers from 1 to 99999 and is described as “Identifiers of Parts”.</td>
</tr>
<tr>
<td>Entities</td>
<td>An entity corresponds to some real world entity or object we would like to store information about. Entities are constructed from a collection of attributes. May also be ‘artificial’, e.g.: an association entity possibly with no real world analogue. Entities have a name, a description and a structure gained by aggregating attributes together. In the supplier/parts example the entity Part is constructed by aggregating the attributes Part_ID, Part_Name and Part_Description. It may be called Part and be described as “the part entity”.</td>
</tr>
</tbody>
</table>
| Relationships| Describe some relationship between one or more entities only. Three types exist distinguishable by their cardinalities: (1 – 1, 1 – M, M – N). All may exist in a Chen E-R Model diagram although the third (M – N) must at some time be decomposed to two (1 –
M) relationships. Unlike other E-R variants, the Chen E-R Model does not have a membership class for relationships.

Up to the physical design stage, a relationship has a name and a description for each permutation of the entities involved in the relationship. It also possesses a cardinality (described above).

Suppliers send shipments of parts. Many parts may be shipped by many suppliers. This (many to many) relationship is characterised by the Shipment association entity. The two derived relationships (each one to many) may be referred to as “Shipped” and “Shipped by” and have appropriate descriptions.

5.5.2 RIPPLE Concepts

Recall the four basic concepts which lie at the heart of RIPPLE. They are:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attributes</td>
<td>A named atomic value drawn from a named value pool (domain). Attributes have a name, description and a domain.</td>
</tr>
<tr>
<td>Domains</td>
<td>A named value pool. Described as either enumerated or constrained base types (e.g.: integers, characters strings etc). Domains have a name, description and the domain definition itself.</td>
</tr>
<tr>
<td>Objects</td>
<td>An abstraction drawn on a set of items stored by RIPPLE. Abstraction mechanisms used are: Aggregation, Generalisation, Subset Hierarchy and Collection. Legal abstraction mechanisms may be specified for an object based concept. Objects have a name, description and a structure definition (governed by the particular abstraction mechanism used to construct it).</td>
</tr>
<tr>
<td>Connections</td>
<td>Allow one or more items stored by RIPPLE to be ‘connected’ to each other. The concepts that may be connected are specified in the method definition. Connections have a name, a description (for the connection as a whole) and record the object class members that participate in the connection.</td>
</tr>
</tbody>
</table>

Note: Every item stored by RIPPLE also has a unique identifier. Names are unique within base concept classes only. This means that names are unique for RIPPLE attributes and the other RIPPLE concepts i.e. that instances of different RIPPLE concepts may have the same name. In addition, further property information may be associated with an item stored by RIPPLE via a property object.
5.5.3 A Comparison between Chen E-R and Ripple concepts

The descriptions of the Chen E-R Model and Ripple concepts allow some conclusions to be drawn about how they should be mapped:

1. Chen E-R Model attributes map to Ripple attributes.
   Modifications: No further modification required.

2. Chen E-R Model domains map to Ripple domains.
   Modifications: No further modification required.

3. Chen E-R Model entities map to Ripple objects.
   Modifications:
   (a) Construction limited to aggregation only. They may include both attributes and composite attributes (which may be represented by Ripple objects but which are differentiated from entities if required).

4. Chen E-R Model relationships map to Ripple connections.
   Modifications:
   (a) Relationship ends must also include a relationship description from the perspective of that end. They also have a cardinality.
   (b) Relationships must only connect entities (this does not include composite attributes).

The following property object is associated with relationship ends. It describes the structure of that property information and resides in the meta-metadata layer. As stated above, each relationship end has a description and a cardinality (one or many). This provides property information for ends of relationships.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description</td>
<td>c250</td>
<td>Holds the description of the relationship from this end 'outwards'.</td>
</tr>
<tr>
<td>Cardinality</td>
<td>il</td>
<td>Contains the cardinality of this end. This attribute has a domain representing the cardinalities (One, Many). Note: In this case the method uses a domain based on one byte integers and will use appropriate values to denote the One and Many cardinalities. Alternatively, a domain could be set up to have the values “One” and “Many” explicitly defined via an enumeration.</td>
</tr>
</tbody>
</table>

Table 5.5.1: Property object describing connection end property information for Chen E-R Model relationships.

The following relations are used to contain concept mapping information such as which abstraction principles may be used to construct an object based method concept. The cm prefix on the relation names is derived from concept mapping. These tables all hold data in the meta-metadata layer.
The description of each of the following relations is self-descriptive via the Description of Purpose field. For a complete set of E-R diagrams representing this structure see Appendix B.

**Relation: cm_connection**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>i4</td>
<td>Identifier of a concept mapping onto a \textit{Ripple connection}. Listed in concept.</td>
</tr>
<tr>
<td>Naryness</td>
<td>i4</td>
<td>Identifier of a domain specifying the legal cardinalities for this connection concept.</td>
</tr>
</tbody>
</table>

Table 5.5.2: Relation containing concept mapping information for relationships as a whole.

**Relation: cm_conn_end**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>i4</td>
<td>Identifier of a concept that maps onto a \textit{Ripple connection end}. Listed in concept.</td>
</tr>
<tr>
<td>EndConcept</td>
<td>i4</td>
<td>Identifier of a concept (not based on connection ends) that may exist attached to this \textit{connection end} concept.</td>
</tr>
</tbody>
</table>

Table 5.5.3: Relation containing concept mapping information on which method concepts may be connected by this connection concept.

The relations cm_connection and cm_conn_end do not store information about which connection end concepts may be associated with which connections. The relation cm_conn_conn_end describes this association.

**Relation: cm_conn_conn_end**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conn_Concept</td>
<td>i4</td>
<td>Identifier of a \textit{connection} based concept. Listed in Concept</td>
</tr>
<tr>
<td>End_Concept</td>
<td>i4</td>
<td>Identifier of a \textit{connection_end} based concept.</td>
</tr>
</tbody>
</table>

Table 5.5.4: Relation containing concept mapping information on which connection end concepts may lie at the end of this connection concept.

Method concepts using \textit{Ripple objects} as their base concept have constraints placed on their abstraction of other method concepts. Each has a set of method concepts that may be involved in each allowed abstraction principle for that concept.

**Relation: cm_object**

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifier</td>
<td>i4</td>
<td>Identifier of a concept that maps onto a \textit{Ripple object}. Listed in concept.</td>
</tr>
<tr>
<td>Ab_Types</td>
<td>i4</td>
<td>Identifier of a \textit{Ripple} domain containing the valid abstraction mechanisms for this object based concept.</td>
</tr>
</tbody>
</table>

Table 5.5.5: Relation containing concept mapping information on objects as a whole.
The table below describes how the concepts in Chen E-R Model map to basic RIFLE concepts. There are three tuples in this table (shown in table 5.5.7).

### Relation: concept

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identifer</td>
<td>i4</td>
<td>The identifier of a concept.</td>
</tr>
<tr>
<td>Name</td>
<td>c50</td>
<td>Concept name.</td>
</tr>
<tr>
<td>Description</td>
<td>c250</td>
<td>Concept description of this mapping from method concept the RIFLE concept.</td>
</tr>
<tr>
<td>Base_Concept</td>
<td>i1</td>
<td>The RIFLE base concept this Chen E-R Model concept is mapped onto.</td>
</tr>
<tr>
<td>POID</td>
<td>i4</td>
<td>Identifier of the property object for this concept.</td>
</tr>
</tbody>
</table>

Table 5.5.7: Relation containing concept mapping information for concepts as a whole.

In this section concepts in the Chen E-R Model and RIFLE conceptual model have been examined. A set of concept mappings between the Chen E-R Model and RIFLE conceptual model concepts has been established. Some extra properties were required, the structure of which is described by the property object shown in table 5.5.1.

Transcription of this information into the RIFLE method definition system gives RIFLE all the information required to provide appropriate storage structures and repository facilities to a Chen E-R Modelling tool. Once these details have been entered into the method definition system then they are hidden from the end user of tools utilising RIFLE.

The following relations show an example of how this works. They contain values similar to those that would actually appear in the core system relations for the simple supplier/parts example used in this section. The table directly below (Relationship End POID) holds instances of the property object associated with ends of the relationship (i.e. this is metadata).
All the following tables contain information that resides in the meta-metadata layer. The information they hold describes the different concepts present in the Chen E-R method and they ways instances of them may be constructed. The IDs present in these tables relate to the IDs in the Concept table below. Their values, including those in the table above have been chosen arbitrarily.

Relation: cm_connection

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Degree</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 (Relationship)</td>
<td>A \textit{Ripple} domain with a value representing an entity.</td>
</tr>
</tbody>
</table>

Relation: cm_conn_end

<table>
<thead>
<tr>
<th>Identifier</th>
<th>End Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 (relationship end)</td>
<td>A value representing an entity.</td>
</tr>
</tbody>
</table>

Relation: cm_conn_conn_end

<table>
<thead>
<tr>
<th>Conn Concept</th>
<th>End Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 (Relationship)</td>
<td>A value representing a relationship end.</td>
</tr>
</tbody>
</table>

Relation: cm_object

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Ab Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 (Entity)</td>
<td>A \textit{Ripple} domain with a value representing \textit{aggregation}.</td>
</tr>
</tbody>
</table>

Relation: cm_object_construct

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Ab type</th>
<th>Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>3 (Entity)</td>
<td>Aggregation</td>
<td>Values representing \textit{attribute} and \textit{entity} concepts.</td>
</tr>
</tbody>
</table>

Relation: concept

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
<th>Base Concept</th>
<th>POID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Chen E-R Attribute</td>
<td>A Chen style attribute</td>
<td>Attribute</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Chen E-R Domain</td>
<td>A Chen style domain</td>
<td>Domain</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Chen E-R Entity</td>
<td>A Chen style entity</td>
<td>Object</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>Chen E-R Relationship</td>
<td>A Chen style relationship</td>
<td>Connection</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>Chen E-R Relationship End</td>
<td>A Chen style relationship end</td>
<td>Connection End</td>
<td>6</td>
</tr>
</tbody>
</table>
5.6 An Actual Method Definition Case Study

Chapter Eight presents a small case study involving the description of several method definitions and a practical demonstration of the RIPPLE system. The method definition presented as an example in section 5.5 is used in the concept demonstration presented in Chapter Eight.
Chapter Six: Semantics

6.1 What are Semantics?

The Collins Concise English Dictionary gives a definition of *semantics* as "... the study of the relationships between signs and symbols and what they represent...". Derived from *semantic* meaning "...of or relating to the meaning of different words or symbols...", semantics are used in many areas. In linguistics it is the study of the meaning of language. In denotational semantics the meaning of programs is the subject of study. In logic they are the principles that determine the truth values of the formulas in a logical system. Semantics are all around us – we may not always recognise them for what they are.

*RIPPLE* also makes use of semantics. Here, semantics is also all about meaning. In the definition of semantics given above, signs and symbols refer to both facts and knowledge (such as attributes and relationships) and the ways those facts or knowledge are manipulated or operated upon. In this sense the term *semantic meaning* of an action or operation refers to the conditions, actions and consequences thereof. In a variant of the ever-popular E-R method, it may be useful to require that to delete an entity, all participating attributes must also be deleted. Thus one of the semantics for the operation *delete an entity* is to ensure all attributes are deleted after the entity is deleted. If this is not possible then the *delete an entity* operation cannot succeed. It is easy to see that this semantic constraint of deleting an entity will result in the semantic constraints for deleting an attribute to also be invoked.

There have been some attempts to deal with the semantics of such situations. One such attempt is the UDL language designed by Date [Date86a]. Of particular importance is the ability to handle situations such as the cascading delete noted above. *RIPPLE* is capable of providing sufficient services to enable such semantic checking to be performed both with regard to the internal core system and external method semantic checks.

As has been mentioned briefly in section 3.7, semantics are both very important and very useful to *RIPPLE*. Section 6.2 discusses why *RIPPLE* needs semantics. Section 6.3 looks at the different types of semantics used. Sections 6.4 and 6.5 discuss *RIPPLE* and method semantics.

6.2 Why does *RIPPLE* need them?

*RIPPLE* not only makes use of semantics, it could not function without them. They guide the operations taking place at every level in the entire *RIPPLE* system. It is imperative that information stored by *RIPPLE* is semantically correct – with respect
to both RIPPLE and the tools using it. Without such an assurance a repository loses effectiveness. It is not the intention for RIPPLE to constrain what a method may do while manipulating its view of a model. However, the integrity of the information stored by RIPPLE must come first.

By their very nature semantics pervade the entirety of RIPPLE. From the core system to the method definition system, to the extensible interface and the tools using RIPPLE, semantics are never far from view. Without semantics RIPPLE could not control the creation of, for instance, E-R entities with inappropriate abstraction principles or stop E-R relationships connecting attributes as well as entities.

6.3 Static & Dynamic Semantics

There are two types of semantics used by RIPPLE. Though both types are in fact still semantics, they have subtly different origins. The first, dubbed static semantics, relate to the semantics governing the actions of the core system. That is the core routines within RIPPLE that deal only with RIPPLE concepts (i.e. attributes, domains, objects and connections). The second, dubbed dynamic semantics, relate to the semantics present within definitions of methods described to RIPPLE via the method definition system. These semantics cannot be known, a priori, by RIPPLE so must be handled in generic and dynamic fashions at the time situations are encountered.

6.4 Static Semantics

These semantics control internal operations within the core system, the extensible interface between RIPPLE and tool applications, and the method definition system dealing with RIPPLE concepts. None of these static semantics are used when dealing with method concepts (see section 6.5). These are called static semantics because they may be enumerated and defined before ever using the system. They could be said to be the semantics of RIPPLE itself. Static semantics may be found in other places – such as those used in the relational model formulated by Codd ([Codd70], [Codd75]).

6.4.1 Method Semantics

These refer to the semantics that apply to actions performed during the process of defining a method to RIPPLE. These include the static semantics described in section 6.5.1 for the RIPPLE core system. In particular, method semantics relate to manipulating the RIPPLE concept concept. The create, read, update and delete operations below all refer to the concept concept.
Create  Creating a concept requires that no other concept be present with that name. The concept may also have a property object associated with it. This property object is a special type of *RIPPLE object* that may only created by aggregating *attributes* and *Object Bin* type objects together. This is a further restriction on the core system static semantics for creating an *object*.

Read  When defining a method, all the concepts present in the current model database may be viewed. This allows the person defining a method definition to browse those concepts present in other method definitions and to import them if useful.

Update  Once a concept has been created only its name and description may be changed. A property object associated with a concept may have attributes added to it. Removal of attributes from a property object is not allowed. This helps to keep information in the repository consistent. The appropriate changes are reflected in structures maintaining property information in the repository.

Delete  No instances of the concept may be present in the model.

6.5 *RIPPLE* Static (Core System) Semantics

6.5.1 Static Core System Checks

This section examines some of the checks that exist in the *RIPPLE* core system. All of these checks are actually implementing static semantic constraints. This is not meant to be a complete list but serves to give an idea of the sort of static semantics that are enforced in the core system.

These semantics are unlikely to be sufficient for a method using *RIPPLE* even when considering the dynamic semantics set up through *Method Definition*. In the core system *delete an attribute* operation further actions could be performed in the *RIPPLE/Application* interface layer (discussed in section 9.4). In this way the tool may delete the domain when it deletes an attribute by including a call to *delete a domain* in the interface layer routine performing *delete an attribute*.

All of these semantic checks operate at the meta-metadata level. Many also operate at the metadata level. When otherwise stated, all semantics described here do apply to all levels in the metadata hierarchy.
6.5.1.1 Common Checks

All the core system concepts have a number of common semantic checks. These are noted here. Those checks which are not common are specified explicitly in the following sections. The term 'current method' refers to the method currently performing operations on information within the repository.

- **Create**
  An item must not already exist in that base concept class with the same name. E.g. if attribute X exists then another attribute called X could not be created, but an entity named X could be created.

- **Read**
  The item must exist in the *inuse* list for the current method. If the item does exist in the repository the current method cannot use it unless it is present in its *inuse* list. If permitted, the current method may import the desired item (adding it to its *inuse* list) and then use it (see Chapter Seven for a detailed discussion of information sharing in RAPPLE).

- **Update**
  Any item may have its name and description changed by any method having that item within its *inuse* list. The new name must not be the same as any other item of the same base concept.

- **Delete**
  To destroy an item in the repository requires that no other method but the current method currently includes that item in its *inuse* list. If other methods are using that item then it is *unlinked* from the current method only. *Unlinking* an item refers to the action of removing an item from the *inuse* list of a method. The item is, however, still present within the repository. If an item is unlinked and only one method is using it then it is automatically destroyed. In a sense this operation is now split into two – *Detach* and *Destroy*.

6.5.1.2 Domain Checks

- **Create**
  Domain definitions are not permitted to be recursive. That is, a domain may not include itself as a *abstract* domain part at any place in its own definition. Thus, when a domain is being created, every part being added to that domain is checked to see it does not, itself, contain the domain being created as an *abstract* domain part.

- **Read**
  Nothing in addition to common checks.
Update  Domain definitions may not be altered as any alteration may make values of attributes using that domain inconsistent with the new definition. Note that domain definitions may be created at the metadata level as no values are stored that depend on them.

Delete  The domain may not be deleted if there exist any attributes using it, or if it is involved in any object or connection definitions. Further, it may not be deleted if it takes part in another domain's definition (as an abstract domain part).

6.5.1.3 Attribute Checks

Create  Nothing in addition to common checks.
Read   Nothing in addition to common checks.
Update Nothing in addition to common checks.
Delete  The attribute must not exist in any object or connection within the current method. The domain used by the attribute is not deleted if the attribute is successfully deleted. Thus it must be separately deleted if this is desired.

6.5.1.4 Object Checks

Create  Nothing in addition to common checks.
Read   Nothing in addition to common checks.
Update Objects are not permitted to be recursive. That is, an object may not contain itself as part of its definition.
Delete  The object must not exist in any object or connection definition in the current method.

6.5.1.5 Connection Checks

Create  Nothing in addition to common checks.
Read   Nothing in addition to common checks.
Update Nothing in addition to common checks.
Delete  The connection must not exist in any object or connection definition in the current method.

6.5.1.6 Concept Checks

These static concept semantics checks apply to the meta-metadata level only.

Create  Nothing in addition to common checks.
Read   Nothing in addition to common checks.
6.5.2 RIPPLE Dynamic Semantics

These semantics also have some, indirect, control over operations in the core system. However their influence appears higher up at the point where method concepts are being manipulated. These are called dynamic semantics because, before running the system they can not be quantified or described. The example presented in section 3.7 is a good example of this type of semantics. There is no way that RIPPLE could know that that particular E-R Model variant would take such actions when an entity is deleted. For it to know this it would have to be told it explicitly, so that it may apply the appropriate actions at the time the event occurs. These semantics are derived at the time the method is defined to RIPPLE. The common checks outlined in section 6.5.1.1 also apply to these checks. Because of the nature of attributes and domains they do not have any dynamic semantic checks that apply to them.

6.5.2.1 Object Checks

- **Create** When creating an object the requested abstraction principle to be used to create it must be consistent with the set of valid abstraction principles for that concept.

- **Read** Nothing in addition to common checks.

- **Update** Any item being added to an object must belong to a concept present in the list of valid concepts that may be added to this particular object concept instance.

- **Delete** Nothing in addition to common checks.

6.5.2.2 Connection Checks

- **Create** Nothing in addition to common checks.

- **Read** Nothing in addition to common checks.

- **Update** Any item being added to a connection must belong to a concept present in the list of valid concepts that may be added to this particular connection concept instance.

- **Delete** The degree of the connection must be consistent with the domain denoting the valid degree values this connection may have. An exception is made when the degree of the connection
is below the minimum value in the degree domain. This allows a method to populate a relationship definition with entities, for example, without having the initial state of one entity involved in the relationship being rejected.

6.6 Importation Semantics

The semantics of importing information from one method into another are discussed in detail in Chapter Seven. Briefly, one method explicitly allows another method to import instances of a particular concept. A number of checks are performed at the time the permission is created to make sure the sharing of that information does not result in inconsistencies or a semantically nonsensical result.
Chapter Seven: Information Sharing

7.1 Why is Information Sharing Useful?
The ability to share design information between methods is extremely useful and very important if robust and accurate designs are to be achieved. From reducing problems of redundant information to allowing tools to make better, well informed, decisions about the problem at hand, shared information is indispensable. Chapter One has already discussed some of the merits of information sharing.

The benefits of sharing design information have not been lost on vendors in the industry. More and more products trumpet their abilities to share information between methods they support. The systems discussed in Chapter Two are but a small fraction of these.

7.2 Ripple Information Sharing Support
This section describes the support provided by Ripple for sharing information between methods. This support is evident at two levels - the metadata level and the meta-metadata level. A description of the modus operandi used to import items is presented. Additionally the structures used to hold import permissions are described. These permissions control sharing of metadata between methods. In section 2.2 it was noted how stand-alone tools must use external file transfer to share information. With the mechanism described in this chapter information in transparently and safely shared without ever leaving the repository.

7.2.1 Import Permission Structures
The relation shown in figure 7.1 contains information used to control importation of instances of concepts from one method into another. It stores pairings between concepts within methods that 'correspond'. I.e. it denotes the method from which a concept may be imported from, the method that will import it, and the local concept the importing method will assign it. This approach supports the idea of 'based on' importation. In this approach we say that the concept in the importing method is like the concept in another method but which also has some other properties.

The approach taken by Ripple regarding information sharing is a regulated one. There is no high level of intelligence built into the core system to allow it to determine whether one method may import an item from another. To implement such a mechanism is very difficult, if not impossible, given the aim of providing repository support to arbitrary methods. Instead, the allowed paths for sharing information between methods are explicitly recorded as part of a methods’
definition. In effect, a method explicitly grants permission to another method to import instances of a particular concept. As part of that permission, the local concept that will be assigned to that item by the method importing it forms part of the permission. This local concept is then associated with the item once it has been imported into the importing method. Figure 7.1 shows how import permissions are granted and received by methods.

![Figure 7.1: Import permissions are explicitly granted by one method, to others. A method, in turn, receives import permissions granted to it.](image)

This seemingly restrictive regime has a number of advantages. No changes are necessary to the core system to enable it to recognise and provide information sharing support for new methods. The ways in which information is shared between methods are known and defined explicitly. It is thus possible to artificially isolate a method from information it might otherwise be able to share. Further, it allows 'pathways' along which shared information must 'flow' to get from one method to another. Figure 7.2 illustrates such a situation. In this figure we see three methods with some import permissions between them. We note that method C has permissions to import items from both other methods. However, a situation could arise where items from method A could legally be imported into method C, but would be more useful if they had to first be imported into method B and from there into method C. If the direct permissions between methods A and C are not created then the only way in which method C can share those items is by importing them off method B. A practical example of this may be a 'pseudo-method' that takes items from one method and converts them into a form more suitable to another, which method then importing those converted items. Such a setup could be more beneficial than to attempt to directly import the items in question.

![Figure 7.2: Pathways along which information may flow between methods may be setup.](image)
Defining import permissions in this explicit manner is both simple and quick to do. The number of permissions that need to be created depends on the number of concepts within the method in question and the number of methods it is prepared to share information with. For a method to import an item it is thus sufficient to check these permissions.

This structure also records correspondences between connection end concepts without the need for any other structures. The relation shown in figure 7.3 shows the structure of the Import Permission entity shown in figure 7.1.

Relation: Import_Permissions

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Type</th>
<th>Description of Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method_From</td>
<td>i4</td>
<td>Identifies the method from which the item may be considered 'being imported from'.</td>
</tr>
<tr>
<td>Method_To</td>
<td>i4</td>
<td>Identifies the method that will import the item.</td>
</tr>
<tr>
<td>Concept_From</td>
<td>i4</td>
<td>Identifier of the concept assigned to the item being imported from the method (Method_From).</td>
</tr>
<tr>
<td>Concept_To</td>
<td>i4</td>
<td>Identifier of the concept the importing method (Method_To) will assign to the item once imported.</td>
</tr>
</tbody>
</table>

Figure 7.3: Relation used to contain importation permissions.

7.2.2 The Importing Algorithm

The process of importing an item into a method is controlled through the RIPPLE core system. A regulated approach is taken (described in section 7.2.1) with allowed importations explicitly defined via the Method Definition system.

To import an item the importing method sends an import request to the core system. This request need not specify the source of the item to be imported. It simply states the desire to share the indicated item. The core system is responsible for resolving from where the item is to be sourced. Within this request is specified the concept it wishes to associate with the imported item once it has been imported. The core system accepts the request and verifies its validity. If it is valid, then that item has the given local concept assigned to it and is appended to the inuse list of the importing method. Figure 7.4 provides a graphical depiction of this process.

When the core system receives an import request it interrogates other methods in the current model searching for relevant import permissions. It retrieves all instances of the given item in all other methods for which there exists a corresponding entry in import_permissions. The core system checks that the concept identifier provided by the importing method matches the concept identifier recorded in the import permission. Only those permissions where the concept identifiers do match are accepted.
An import permission contains four parts (shown in figure 7.3). For a permission to be recorded by the method definitions system the following must hold:

\(\text{Method}_{\text{From}} \rightarrow \text{Method}_{\text{To}} \rightarrow \text{Concept}_{\text{From}} \rightarrow \text{Concept}_{\text{To}}\)

1. **Method\_From** → Identifier of a method from which an item may be imported.
2. **Method\_To** → Identifier of the method that will perform the importation.
3. **Concept\_From** → The concept assigned to the item in the **Method\_From** method.
4. **Concept\_To** → The candidate local concept proposed for the item by **Method\_To** when the importation is requested.

If one or more permissions exist where these conditions are met then the core system arbitrarily picks the first one and applies the importation. Picking a permission arbitrarily from the list is not as bad as it sounds. To verify the legality of an importation, it is sufficient to ensure there exists at least one permission allowing it. In this case where the item is in use by a number of other methods all of which are prepared to allow the requesting method to import it.

The one, slight, exception to this rule occurs with respect to connections. If an item with a connection based concept is imported the core system must also ensure the ends of the imported connection are represented in the importing method. If there exist insufficient permissions to allow all ends of the connection to be imported with the connection body, then the importation is vetoed until sufficient...
permissions exist. To this end *Import Permissions* is again inspected. This time each connection end is imported and assigned the specified local connection end concept in the same fashion as other concepts.

Note: Items existing at the ends of the imported connection are not, in themselves, imported. Instead the references to those items are created with appropriate local concepts. The importing method is free to browse those items but is not permitted to modify them. The importation of a domain, object or connection into a method does not automatically allow modification of any items comprising its structure. In order to do this, the component items must also be imported, if applicable, before they may be modified.

### 7.2.3 Import Permissions and Method Libraries

It is useful to note that the method referenced by *method_to* above, need not currently exist within the model database. It must, however, exist at the time the import permission is created. This further aids the maintenance of a comprehensive library of method definitions. Within this library, each method can set up the appropriate import permissions for other methods in the method library. When a model database is created the required method definitions may be extracted from the method library. These may then be inserted into the method library within the model database.

Before addition of method D

<table>
<thead>
<tr>
<th>Method A</th>
<th>Method B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method C</td>
<td></td>
</tr>
</tbody>
</table>

After addition of method D

<table>
<thead>
<tr>
<th>Method A</th>
<th>Method B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method C</td>
<td>Method D</td>
</tr>
</tbody>
</table>

![Figure 7.5: Addition of an extra method from the method library activates import permissions from already existing methods, and adds some of its own.](image)

The methods in the model database method library will have, possibly many, import permissions that relate to methods not present. These permissions are of no use in the absence of the other methods. However, at a later date another method may be extracted from the method library and inserted into the models' method library (shown in figure 7.5). Now, with no further configuration, the new method is able to share a range of information supported by other methods present in the model.
7.2.4 Method Definition

A method's definition contains importation permissions describing how instances of concepts within it may be shared by other methods. But information sharing is not just useful in the metadata layer. It is also useful within the meta-metadata layer. Here, the actual concepts, and the structures and pieces used to create them, may be shared by other methods. This not only aids the definition of new methods to RIPPLE, but also in the creation of 'roll your own' methods.

At the meta-metadata level there is no need for import permissions. In a sense these are embedded within the RIPPLE method definition system. The reason that rules may be used for controlling sharing of meta-metadata but not for metadata is that method definitions are all described in one 'super-method'. This 'super-method' is RIPPLE. In this sense RIPPLE does indeed act as a method. It has its own concepts and structures and its own set of semantics relating to their manipulation. The RIPPLE 'super-method' could even be described by the RIPPLE method definition system. The same internal mechanisms are used to achieve information sharing at both levels. The main exception is that importations at the meta-metadata level short circuit the permissions test. This is possible because there is only one set of concepts to deal with – the RIPPLE concepts: domains, attributes, objects and connections.

The ability to share information at the meta-metadata layer has useful advantages. It is the prime means by which property information recorded for an item in one method may also be imported into another method along with that item. An item being imported into another method is unlikely to have exactly the same property attributes in the importing method. It is likely, however, that there will be some overlap. The property attributes present in this area of overlap should have exactly the same values in both methods. If they are different the old problems with redundant information come back to haunt us. This will only happen if these property attributes are different (that is, the two attributes have separate definitions and domains). To ensure they are the same they are created in the definition of one method, and imported into subsequent method definitions that also use those property attributes.

A simple example may be derived from the common practice of recording the identity of the person who created an item (attribute, entity, data flow etc, etc...) Thus, when an attribute is created in the Chen E-R Model the creators' identity is a property of that attribute. If this attribute is then imported into a Data Flow Diagram method then the identity of the creator could also be a property of that attribute in that method. If it is not then there is simply no record of its creator. If such a
property is present, and it has been imported from the Chen E-R Model it will contain the same value. If it is not the same property attribute, but a different one with the same purpose, it will remain a null value and represents a redundant copy of its creator (which is already different). Another possibility is to have two creator attributes, \textit{ER\_creator} and \textit{DFD\_creator} so that independent values for these creators can be maintained. Figure 7.6 shows how sharing common property attributes can be used.

![Diagram showing the sharing of meta-metadata](image)

Figure 7.6: A graphical depiction of how sharing meta-metadata can aid information sharing at the metadata level.

Figure 7.3 describes a structure used to hold import permissions. In figure 7.6 above we see a Data Flow method importing an attribute (an instance of the E-R attribute concept) from an E-R method. Figure 7.7 shows a tuple of the table in figure 7.3 containing an import permission allowing this sharing.

<table>
<thead>
<tr>
<th>Method From</th>
<th>Method To</th>
<th>Concept From</th>
<th>Concept To</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity-Relationship</td>
<td>Data Flow Diagram</td>
<td>E-R Attribute</td>
<td>Data Flow Attribute</td>
</tr>
</tbody>
</table>

Figure 7.7: A tuple in the \textit{import\_permissions} relation allowing a Data Flow method to import an attribute from an Entity-Relationship method.

### 7.2.5 Import Permission Creation Constraints

It makes little sense to allow import permissions to be created with abandon. A barrier to the creation of import permissions lies in the following restriction. Permissions may only relate concepts derived from the same \textit{RIPPLE} base concept. For example, method \textit{A} may not import an instance of an \textit{object} based concept from
method B and assign a connection based concept to it. This would correspond to an attempt to import an entity from an E-R method into a Data Flow method and call it a process. This clearly makes little sense and should not be allowed to happen. The Data Flow method importing the entity from the E-R method has to assign an object based concept to it—a data flow data concept perhaps.

Some, further, constraints are placed on creating import permissions. These chiefly relate to the structuring of the concepts involved and concern the Ripple object and connection concepts. This interference is not all bad. In the main it aids in barring semantically meaningless permissions to be created. Attribute and domain based concepts have no constraints on creation of import permissions for them bar that mentioned above. Attributes and domains in the Ripple conceptual model do not have internal structure as do objects and connections. Objects are created using an abstraction principle that denotes whether the components of the object are aggregated together, or form a generalisation etc. Connections have an degree domain that controls the allowed number of ‘ends’ the connection may legally have. These factors interfere with importation of instances of these concepts into other methods.

For object based concepts the two concepts must contain the same sets of abstraction principles used to construct them. This is accomplished via a domain containing the abstraction principles referenced by both concepts. This domain is then referenced by both concepts. Thus comparison of the domain identifiers is sufficient to ensure this condition is met.

For connection based concepts we need to make sure that the degree of the two connection concepts are the same. Importing a relationship from an E-R method allowing ternary relationships into a binary relationship E-R method makes little semantic sense to the binary relationship E-R method. This is also accomplished via a domain containing the valid degree values. This domain is referenced by both concepts. Thus a comparison of domain identifiers is then sufficient to ensure this condition is met.

These constraints may seem restrictive and in some ways they are. For instance, if E-R method A has only binary relationships then an E-R method allowing n-ary relationships should be able to import a binary relationship from method A. However, it is difficult for the core system to infer such situations, especially when such importations may have to be ‘undone’. The reason this may have to happen lies with the second method (B). Its ability to have n-ary relationships means it could modify the relationship it imported from method A into, for instance, a ternary relationship. This will, obviously, be a source of consternation in the
binary relationship method when one of its relationships ceases to be binary and
becomes a ternary relationship.

There is a simple way to avoid this problem. Method B could have a separate
relationship concept that allowed only binary relationships. This concept could
share the same relationship end concepts used by the n-ary relationship concept.
With this new relationship concept it could freely import binary relationships from
method A. Ripple can then correctly police use of the imported relationship in
Method B. Method A is now safe from semantically meaningless modification of
this relationship by Method B. This change may be quite transparent to the tool
application involved. The only changes that need to be made are in the tool
application/Ripple interface (described in section 9.4).
Chapter Eight: R\textsc{ipple} Concept Demonstration

8.1 Introduction

In preceding chapters topics ranging from the motivation behind the R\textsc{ipple} project to details of its conceptual model and the Method Definition system have been discussed. However, it is not sufficient to simply describe R\textsc{ipple}. It is necessary to demonstrate that the theory outlined is more than just rhetoric. A demonstration is required to show the theory amounts to an idea that works.

This chapter discusses Toolset, a simple integrated tool application and the Method Definition system. Toolset demonstrates the soundness of the theoretical basis. It implements a small set of simple design methods. Those chosen are the Chen E-R Model [Chen76], the Teorey, Yang and Fry Extended E-R model [Teor86] and the Gane & Sarson Data Flow method [Gane79]. Together these methods represent both data, and process oriented modelling techniques. All have existed for a considerable time and are well understood.

8.2 The Demonstration Programme

Toolset provides the platform for demonstrating ideas fundamental to R\textsc{ipple} such as information sharing, and mediation of modelling tool interactions. However, these are just two areas in R\textsc{ipple} among many that must be considered:

- The conceptual model that underlies R\textsc{ipple}. This provides the basis for the entire repository management system. In a way this model defines the repository. Does it provide sufficient building blocks and operations that are used to manipulate them? Does it provide the required flexibility and extensibility? These are required to support repository requirements of not only varied methods, but also R\textsc{ipple} itself.

- The R\textsc{ipple} core system. This is the repository engine performing and providing the repository services used by all other sections of R\textsc{ipple}. Is the range of services provided by the core system sufficient to allow other sections of R\textsc{ipple} unhindered functionality? Is the core system flexible and extensible enough to support the requirements demanded of it?

- The Method Definition system. Utilising services provided by the core system, this is the means by which R\textsc{ipple} becomes acquainted with a method. The abilities of this system will have a significant impact on the quality of a method definition. Does it allow us to describe the methods we wish to use in Toolset and other applications? Furthermore, is the quality of these descriptions sufficient to encourage use of this system?
• Information sharing. A repository managed by RIPPLE is a shared store of modelling information. Information may be shared by more than one tool at the same time. Are the services provided through the core system and using method definition information sufficient to provide a robust information sharing capability?

• Interaction Mediation. Tool interactions are mediated by RIPPLE. Undesirable interactions are detected and vetoed. These interactions are manifested via information sharing and the control and modification of this information. Are the services and controls provided through the core system and using information present in the method definitions sufficient to mediate interactions?

• The RIPPLE Public Tool Interface (PTI). Toolset is a client application of RIPPLE. That is, RIPPLE acts as a repository server running on some computer. Toolset may run on (another) computer and initiate contact with the server. The RIPPLE server will, if appropriate, accept Toolset as a client and begin executing requests. From this point every operation Toolset performs on the demonstration model is effected through the RIPPLE Public Tool Interface. Does the RIPPLE PTI provide enough access to repository services to allow Toolset to do what it requires.

8.3 Demonstrating the RIPPLE Conceptual Model

The conceptual model underlying RIPPLE (described in Chapter Three) is demonstrated in a number of places. Directly, it affects all parts of RIPPLE. Initially, methods must be described to RIPPLE via the Method Definition system (described in Chapter Five). This system provides the most direct view of the conceptual model in action. It makes particular use of the conceptual model in its role as a method definer. When working within this system, parts of the conceptual model are always in view, being manipulated and thus may be examined. In this way it is easy to get a good feel for its structure.

Examining the elements of the conceptual model is not, however, sufficient to show two very important properties – flexibility, and extensibility. Sufficient flexibility to support the definition of a variety of methods and an extensibility mechanism allowing extra properties of concepts are very important features in RIPPLE.

8.4 Demonstrating the RIPPLE Core, and Method Definition Systems

Demonstration of the core system is both easy, and very hard, to do properly. It is so fundamental to the entire repository system that any flaws or problems make
themselves apparent very quickly. This same property also tends to make explicit demonstration of the core system difficult. If all areas perform their tasks correctly then the core system may be taken as demonstrated. The RIFFLE Public Tool Interface is similarly hard to directly demonstrate. However, the unhindered functioning of Toolset is sufficient to demonstrate this.

The Method Definition system lends itself more readily to demonstration. The Toolset application must first have methods it supports described using Method Definition before it may use any repository services. In section 8.6.1 the E-R variants supported by Toolset are described and in section 8.6.2 the Data Flow method is described. This exercise provides an excellent demonstration of the method defining system. Even though only simple examples of data and process oriented methods are featured, it is easy to see that Method Definition is quite capable of describing many other methods.

8.5 Demonstrating Information Sharing & Interaction Mediation

Toolset supports three methods – two E-R variants and a Data Flow variant. There are a number of ways information may be shared between them. The most obvious being the sharing of attributes, entities and relationships between the Entity-Relationship methods. Additionally, attribute and domain information, for example, may be shared between the Data Flow and E-R methods. Method Definition allows permissions for sharing data between methods to be set up (described in Chapter Seven). Import permissions are explicitly granted by one method to another creating a permissions list. This list similar to a capability table used by some operating systems such as VMS.

Through this shared data, methods and tools interact with one another. Though this interaction takes place via the core system there are nevertheless many opportunities for inconsistencies to arise unless prevented. The consequences of non-mediation of these processes are outlined in Chapter One.

8.6 Describing the Demonstration Methods to Method Definition

This section shows the process of describing methods to be used in the demonstration of the Toolset application to Method Definition. The first section covers the Chen E-R and Extended E-R methods. The second section covers the Data Flow method. In each section a series of screen snaps has been included to illustrate all aspects of describing a method to Method Definition. Some assumptions and concessions have been made about some aspects of the method definitions in order to keep them small and simple. Thus, while lacking in fine detail and finesse they embody all the
necessary features to allow a complete demonstration of both the Method Definition system and the Toolset application (demonstrated in section 8.9).

8.6.1 Describing of E-R Variant Methods

The Chen E-R and Extended E-R Method variants to be described to Method Definition are outlined in section 3.2 and will not be repeated here. It is impractical to show the creation of every concept in the two E-R method variants here explicitly. Instead, a sample of these concepts are defined. Tables 8.1 and 8.2 describe the concepts and their properties as they will be described to Method Definition.

Chen E-R Method:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Supported Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Type: Key, non-key</td>
</tr>
<tr>
<td>Domain</td>
<td>No extra properties.</td>
</tr>
<tr>
<td>Entity</td>
<td>No extra properties.</td>
</tr>
<tr>
<td>Relationship</td>
<td>No extra properties.</td>
</tr>
<tr>
<td>Relationship End</td>
<td>Description from end.</td>
</tr>
<tr>
<td></td>
<td>End cardinality.</td>
</tr>
<tr>
<td></td>
<td>List of attributes that are keys in the entity attached to this end of the relationship.</td>
</tr>
</tbody>
</table>

Table 8.1: Properties of concepts in the Chen E-R method.

Theory, Yang and Fry Extended E-R Method:

<table>
<thead>
<tr>
<th>Concept</th>
<th>Supported Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attribute</td>
<td>Type: Key, Non-Key</td>
</tr>
<tr>
<td>Domain</td>
<td>No extra properties.</td>
</tr>
<tr>
<td>Entity</td>
<td>No extra properties.</td>
</tr>
<tr>
<td>Relationship</td>
<td>No extra properties.</td>
</tr>
<tr>
<td>Relationship End</td>
<td>Description from end.</td>
</tr>
<tr>
<td></td>
<td>End cardinality.</td>
</tr>
<tr>
<td></td>
<td>Membership class.</td>
</tr>
<tr>
<td></td>
<td>List of attributes that are keys in the entity attached to this end of the relationship.</td>
</tr>
</tbody>
</table>

Table 8.2: Properties of concepts in the Extended E-R method.

The Method Definition and Toolset applications have been developed using the Ingres forms system. These forms allow single fields and scrollable tables to be created along with 'trim' text that acts as a backdrop to the fields and tables. At the bottom of each form is a ring menu from which the user may choose options. To the left and right of this line of menu options there may be ‘<’ and ‘>’ characters. These indicate that the ring menu options continue to the left and right past the limits of the screen.

Figure 8.1 shows the screen presented to the user when Method Definition is run. It shows the current model in use and the list of all methods defined in this model.
This snap represents the state of the model after all three methods have been defined. The RIPPLE System Administration method entry shown is a 'method' created on initialisation of a model database. It provides a place for RIPPLE to store a small number of items useful to its operation.

Initially, a method must be created before any part of it may be defined. Figure 8.2 shows details of the Extended E-R method after it has been created. Once created, a method acts as a shell which is then populated by concepts in the method as they are defined.

This remainder of this section describes the creation of the Chen E-R relationship concept. This requires the need to create a property object for properties of the relationship's ends. This, in turn, requires that attributes for those properties be
created. They, in turn, require domains to be defined before they may be created. As can be seen, a range of RITPLE concepts will ultimately be involved in creating this relationship concept. Each of these steps is now taken (in reverse) to arrive at the finished Chen E-R relationship concept.

From Table 8.1 we see that ends of relationships have three properties (a description from the perspective of that end, an end cardinality and a list of key attributes). Only the End cardinality property definition is explicitly shown here. Figure 8.3 shows the simple domain part created to hold the legal cardinality values. An enumeration of the values \{1, 2\} or \{Key, Non-Key\} could also have been used.

![Simple Domain Part Editing](image)

**Figure 8.3:** Simple domain part containing legal values for relationship end cardinality.

The domain for the relationship end cardinality is then created. The simple domain part shown in Figure 8.3 is then added to it. Figure 8.4 shows the resulting domain. This domain may now be used in the construction of an attribute.

- 94 -
Domain Definition

Name: Cardinality values domain
Description: Cardinality value for the connection between the relationship and an entity.
Operator: Intersection
Consistent Domain Parts: Yes

Component domains and domain parts

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Op</th>
<th>Con</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CER Cardinality Valu</td>
<td>Values for relationship cardinality. Legal</td>
<td>*</td>
<td>***</td>
<td>Part</td>
</tr>
</tbody>
</table>

Figure 8.4: Relationship end cardinality showing inclusion of relationship end cardinality simple domain part describing the actual values.

Once the domain has been created the attribute that will be used to hold the value for the cardinality of a relationship end may be created. Figure 8.5 shows the resulting attribute.

Attribute Definition

Name: End Cardinality
Description: Cardinality of the connection between a relationship end and an entity.

Domain: Cardinality values domain

Figure 8.5: Property attribute for relationship end cardinality.

At this point we have defined one of the properties of an end of a Chen E-R relationship concept. The other two property attributes are created similarly. Once these property attributes are created we are ready to construct the relationship end property object. Figure 8.6 shows this property object after it has been created and the properties added to it.
The highlighted row in the list of components in figure 8.6 denotes an Object Bin property. This property allows a list of items to be stored. Such a list could detail attributes acting as keys in the entity connected to an end of the relationship. Figure 8.7 shows the structure of this property obtained by choosing Browse Next from the menu in figure 8.6.

Now that we have defined all properties that a Chen E-R relationship needs for its ends, this concept may be created. Figure 8.8 shows the creation of the relationship concept. Note that the Chen E-R relationship concept itself has no properties. As mentioned previously, the created properties apply only to the ends of the relationship. Other types of connection could have their own properties.
The degree domain shown in figure 8.8 is a domain containing legal values for the degree of this connection concept. In this case the domain has a single value in it: 2. This indicates that only binary relationships are permitted (see table 8.1). Once the relationship concept is created, the valid Chen E-R concepts that a relationship may connect need to be defined. Figure 8.9 shows the description of the relationship end for the Chen E-R relationship concept. This is not a concept in the usual sense used in this thesis, but shares a similar structure (such as having properties).

The Chen E-R relationship concept has now been created. During this process we have seen how domains, attributes and property objects are created at the meta-metadata level. The creation of no further Chen E-R concepts will be shown here. Examples of other types of concepts may be found in the definitions of the Extended E-R and Data Flow methods later.
The permissions allowing other methods to import instances of concepts from this method also need to be defined. Figure 8.10 shows the creation of a permission allowing the Data Flow method to import Chen E-R attributes. These permissions suggest the ways tools will eventually interact. The boxes and arrows form a collage with the actual screen snap (in behind) showing a sequence of pop-up windows providing choices for the current method concept, importing method and importing method concept respectively.

![Figure 8.10: Creation of an import permission](image)

We now move to the definition of the Extended E-R method. This is very similar to the definition of the Chen E-R method. The main difference is that Extended E-R entities may be involved in generalisations and subset hierarchies and that relationships are not restricted to being binary. Additionally, some extra properties are supported for some concepts. The definition of the Extended E-R method benefits from the prior definition of the Chen E-R method. For example, the domains Description from end and End cardinality relationship end properties are shared removing the need to define those domains again in the Extended E-R method.

The creation of an Extended E-R entity concept to illustrate aspects of object based method concepts is all that will be shown here. The Extended E-R method allows more complex entity types to be modelled than the Chen E-R method. In addition to the work horse aggregation abstraction principle, generalisations and subset hierarchies may also be used.

Figure 8.11 shows the definition of this concept. Once created, the ways in which this object based concept may be constructed must also be defined. The two boxes labelled Abstraction Types and Abstractable Concepts in figure 8.11 show this information. Each abstraction type, listed in the left hand box, has a list of method
concepts in the right hand box that may be involved in its construction using that abstraction mechanism. In this case an Extended E-R entity constructed using the generalisation abstraction mechanism may only use Extended E-R entities in its construction. Appropriate entries for the aggregation and subset hierarchy mechanisms are also present. In this way R/PPLE can veto an attempt to add an entity, for instance, to an aggregation involving attributes only.

Object Based Concept Definition

<table>
<thead>
<tr>
<th>Concept Name:</th>
<th>Extended E-R entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>POID:</td>
<td>No Properties</td>
</tr>
<tr>
<td>Description:</td>
<td>Describes Extended E-R entity concept</td>
</tr>
</tbody>
</table>

Abstraction Types Domain: Extended Entity Abstraction Types

<table>
<thead>
<tr>
<th>Abstraction Types</th>
<th>Abstractable Concepts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aggregation</td>
<td>Extended E-R entity</td>
</tr>
</tbody>
</table>

Figure 8.11: Creation of an Extended E-R entity concept showing ways in which this concept may be constructed.

8.6.2 Description of Data Flow Method

The Gane & Sarson Data Flow Method variant is outlined in section 3.2 and will not be repeated here. It is impractical to show the creation of every concept for the Data Flow method here explicitly. Instead, a sample of concepts will be defined that have not been previously covered. Table 8.3 describes the concepts and their properties as they will be described to Method Definition.

<table>
<thead>
<tr>
<th>Concept</th>
<th>Supported Properties</th>
</tr>
</thead>
</table>
| Process            | Number
|                    | A list of sub-processes within this process.              |
| External entity    | Nature: Source or sink.                                   |
| File store         | No extra properties.                                      |
| Data flow          | Source process
|                    | Destination process
|                    | List of data items flowing along flow.                    |
| Data flow end      | Description from end.                                    |

Table 8.3: Properties of concepts in the Data Flow method.
Data Flow methods have a number of concepts that map onto the \texttt{RIPPLE} object concept (discussed in section 5.2.3.1). Some of these concepts (namely \textit{external entities}, \textit{processes} and \textit{file stores}) do not have internal structure in the manner of an Entity-Relationship entity but do have properties (such as \textit{name}, \textit{process number} etc). Figure 8.12 shows the creation of a Data Flow \textit{external entity} concept. Note that although Data Flow external entities may be created using aggregation this does not require that items must be added to it. This is attained by not specifying any concepts that may be used to build it. The blank \textit{Abstractable concepts} table in the screen snap shown in figure 8.12 illustrates this.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{figure8.12.png}
\caption{Creation of Data Flow \textit{external entity} concept.}
\end{figure}

The Data Flow concepts \textit{process}, \textit{file store} and \textit{external entity} may all be connected by data flows. These concepts are all created in a similar manner to that shown in figure 8.8. Figure 8.13 shows the resultant description for the ends of a data flow. The values present show that a data flow could be created between a file store and an external entity (which is illegal in the Data Flow method). This problem may be partially rectified by creating two separate data flow concepts. A natural way to represent this is to create a data flow concept that connects external entities and processes and another that connects file stores and processes. However, it cannot stop a file store from being connected to another file store. This is an example of the more complex semantic checks \texttt{RIPPLE} itself is not able to perform. Complex semantic checks like this should be included in the semantics applied by the application. Section 9.4.2 further discusses the role of application semantic checks. Section 10.4 includes this area as one to that could be expanded upon in \texttt{RIPPLE}. 

- 100 -
In the Gane and Sarson Data Flow Diagram method levelling may be used to reduce complexity of Data Flow Diagrams by replacing smaller diagram fragments with higher level processes. RUP can support this technique using Object Bin properties. All processes, data flows etc in a sub-process may be placed into such a property which would form part of the property object for the process concept. This simple yet very powerful feature of RUP’s concept extensibility mechanism automatically solves integrity problems such as ensuring consistency of incoming and outgoing flows.

8.7 Sharing Information Between Methods

Now that the Entity Relationship method variants and the Data Flow method have been described we can define the ways these methods may share information among themselves. For completeness, information sharing between both the data and process oriented methods will be shown. In section 8.6.1 import permissions were created in the Chen E-R method allowing the other methods to share metadata information stored by it. Figure 8.14 shows the complete set of permissions created in the Chen E-R method. Figures C.5 and C.10 in Appendix C show all import permissions created in the Extended E-R and Data Flow methods respectively.
It may be seen from this list that the Extended E-R method imports instances of a variety of different concepts from the Chen E-R method and the Data Flow method imports attributes from it. The reader will appreciate there are a number of extra permissions that could be included in this list such as domain sharing between the other two methods.

8.8 The Demonstration Model

As mentioned above, the simple Supplier/Parts model used for this demonstration follows a theme used throughout the thesis. This section describes a simple Supplier/Parts model that aims to show as many aspects of RIPPLE as possible while remaining simple and understandable.

Figures 8.15, 8.16 and 8.17 show graphical depictions of the different views of this model by the different methods. Comparing the two E-R variant diagrams shows the difference between approaches that may be taken to model employee sub-types. The Chen E-R method must either record all the details in a single entity, or have relationships between the Employee, Manager and Worker entities (as shown in figure 8.15). This approach is inadequate as it fails to model the fact that managers and workers are actually employee sub-types. Additionally, employees cannot be both workers and managers so these entities must have optional membership classes in these relationships. Unfortunately, this permits an Employee to be neither a Worker or a Manager, or to be both! This situation is neatly resolved by the Extended E-R method where a generalisation is used to model the fact workers and managers are sub-types of employees.
The diagram shown in figure 8.16 is very similar to the Chen E-R model diagram in figure 8.15 but for one difference. The two relationships between the Employee, Manager and Worker entities are now replaced via a generalisation. This correctly models the Worker and Manager sub-types of the Employee entity.

Figure 8.17 shows the process of adding a part to the parts database. Initially a data entry clerk enters details of the part to be created. These details are then validated with an appropriate error message being passed back to the clerk if the validation fails. If these details are correct the new part is assigned a unique part number and then created. Finally, the status of this operation and the number of the new part are relayed back to the data entry clerk.
8.9 Creating the Demonstration Model

This section outlines the description of some items in the demonstration model. In the Chen E-R model view, the Employee entity and the Supplier employs employees relationship are created. In the Extended E-R model view the generalisation the Employee entity takes part in with the Worker and Manager sub-types is created. Finally, in the Data Flow model view the data flow from the Clerk external entity to the Validate part details process is created.

A similar sequence of steps to those taken in the definition of the Chen E-R relationship concept will be followed here. Figure 8.18 shows the screen presented to the user when Toolset is run. It shows a list of all the methods currently in use manipulating the model. The user may move into any one of these methods (bar the RIPPLE System Administration method) and manipulate the information stored within it.
8.9.1 Creating the E-R Variant Model Views
Here, the generalisation used to model the Employee to Worker, Manager sub-typing is created. Figure 8.19 shows the enumerated domain part denoting the values the attribute Employee type can contain – ‘Worker’ or ‘Manager’.

An Extended E-R domain is then created and has the domain part shown in figure 8.19 added to it. The resulting domain has a structure similar to that shown in figure 8.5 and has the properties shown in figure 8.20. Only name and description are listed as properties due to the No properties property object being given to this concept.
This domain is then used in creating an Extended E-R attribute — *Employee type* — which also has no extra properties (just name and description). This attribute is used to distinguish employees between managers and workers. The Extended E-R entity for *Employee* is then created and *Employee type* and other attributes are added to it. Figure 8.21 shows the structure of this entity. This entity, like the attribute shown above, has no properties of its own other than its name and description. The same applies to the *Supplier employs employees* relationship created below in figure 8.21.

With the creation of the *Employee* entity the *Supplier employs employees* relationship may then be created between the *Employee* and *Supplier* entities. Figure 8.22 shows the items participating in this relationship.
Each of the participants at the ends of the Supplier employs employees relationship have properties of their own which may also be viewed and edited. Figure 8.23 shows the properties of the attachment between the Employee entity and the Supplier employs employees relationship. The value '2' for End cardinality indicates a Many cardinality (see table 8.2). See figures 8.3 and 8.4 for a description of the End Cardinalities domain.

One of the properties shown in figure 8.23, CER Relationship Keys, holds a list of attributes acting as keys in the Employee entity with respect to its involvement with the Supplier employs employees relationship. In this case, the attribute E_Number exists within the list as it is the only key attribute in the Employee entity.

Earlier, a problem with the Chen E-R method’s inability to model sub-types of employees was shown to be solved using a generalisation in the Extended E-R
method. Figure 8.24 shows the object created to hold the sub-types of employees. The reader will notice it is the same as that used to describe the Employee entity created in figure 8.21. One small change is the inclusion into its description a reference to the attribute (Employee type) acting as the 'decider' for the generalisation. This is automatically added and does not form part of the description entered by the user when creating the generalisation entity.

![Figure 8.24: Generalisation object containing sub-types of the Employee entity.](image)

8.9.2 Creating the Data Flow Diagram Model View

Figure 8.25 shows the creation of the Validate part details process. This is created in exactly the same way as the E-R Relationships shown above. The process has no components listed in the components table simply because being a process, it requires none. The same applies to instances of the external entity and file store concepts.

![Figure 8.25: Creation of Validate part details process.](image)
Though processes do not have internal structure in the manner of E-R entities, they still have properties. Figure 8.26 shows the properties of the Validate part details process. As the Validate part details process has no sub-processes of its own, the DFD subprocesses property shown in figure 8.26 contains no processes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFD Process Number</td>
<td>Validate Part Details</td>
</tr>
<tr>
<td>DFD subprocesses</td>
<td>Validates details of part typed in by data entry clerk to validation process</td>
</tr>
</tbody>
</table>

Figure 8.26: Properties of processes

The other processes in the Data Flow view of the model are created similarly as are the Clerk external entity and the Parts file store. With the creation of these items, data flows between them may be specified. Figure 8.27 shows the creation of the data flow between the Validate part details process and the Clerk external entity.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clerk/Validate Part</td>
<td>Data entry clerk</td>
<td>Object</td>
</tr>
<tr>
<td>Validate Part Detail</td>
<td>Validates details of part typed in by data entry clerk to validation process</td>
<td>Object</td>
</tr>
</tbody>
</table>

Figure 8.27: Data flow between Validate part details and Clerk.
The data flow shown in figure 8.27 has some properties of its own. One of the properties of data flows is a description of the data flowing along them. This data is stored in an object bin property. The description of the data flowing along this flow is shown in figure 8.28. All the attributes shown have been imported from the Chen E-R method.

![Figure 8.28: Data flowing along the data flow](image)

Figure 8.28: Data flowing along the data flow between Clerk and Validate part details

The data flow between the Clerk external entity and the Validate part details process has been successfully created. The other data flows in this view of the model are created similarly.

8.10 Summary

In this chapter many parts of the Ripple repository system have been demonstrated. The Ripple Public Tool Interface and core systems have provided the services required for Method Definition and Toolset to do their jobs. The three methods were successfully defined and a simple data model was described to Ripple on the basis of these methods. Information was shared between methods at both the metadata and meta-metadata levels. The extensibility mechanism was heavily used to provide properties such as Relationship end cardinality for relationships in the E-R variant methods. Interaction mediation was working at all times to prevent one method from performing illegal operations on items shared with other methods. The high level of shared information between the three methods used indicates how valuable this is. The controls set up in the Method Definition system and the creation of import permissions by the methods themselves are the chief sources of information for the interaction mediation control system.
Toolset allowed data to be created and manipulated in the different model views seen by the methods at work upon it. At all times Toolset used Ripple's Public Tool Interface to communicate requests and receive answers thus showing its completeness.

Two methods used in this demonstration are very similar. These are the Chen E-R and Teorey, Yang and Fry Extended E-R methods. When defining the Chen E-R method, a number of, sometimes complex, structures had to be created. Following this, the Extended E-R method was defined to Method Definition. Due to the similarity of the two methods the Extended E-R method was able to share parts of the Chen E-R method definition, thus reducing the work required to define it. Later, when Toolset was being demonstrated, this similarity and the sharing of parts of the Chen E-R method definition by the Extended E-R method allowed most of the demonstration model to be directly imported into it. The only items not imported into the Extended E-R view of the demonstration model were the two relationships between the Employee entity and its sub-types (Worker and Manager).
Chapter Nine: \textit{RIPPLE} Implementation Details

9.1 Introduction

This chapter aims to provide some of the details regarding the implementation of \textit{RIPPLE}. The structures of some base concepts in \textit{RIPPLE} are briefly discussed. The design and implementation of the \textit{RIPPLE} application interface (or public tool interface) is discussed in section 9.4. Section 9.5 provides a simple example of how QUEL is embedded into the C source code. A detailed set of specifications and implementation considerations have not been included.

9.2 The Domain System

Some details regarding implementing domains have been discussed in section 3.6.1. Essentially a domain is a tag for a collection of domain parts forming the definition of a value pool. Only the domain parts themselves have any real substance in terms of defining values within the domain. That is, the domain parts define values, and the domain itself controls how those parts are combined together to form the finished result. Figure B.2.1 in Appendix B shows how domain parts and domains are related and their relationship with attributes.

9.3 Objects and Connections

These two core system concepts share similar structures in their implementations. Objects are essentially a collection of items tagged with a particular abstraction mechanism. An extended E-R diagram fragment in appendix B.2.3 shows the structure of objects in the core system and the way they relate to the rest of the conceptual model.

Connections, in a similar fashion, are a collection of items each of which is taken to occupy one end of the connection. Self-relationships are recorded by recording the entity twice in this list. An extended E-R diagram fragment in appendix B.2.4 shows the structure of connections in the core system and the way they relate to the rest of the conceptual model.

Appendix B contains a full set of extended E-R diagrams showing the structure of the \textit{RIPPLE} core system.

9.4 \textit{RIPPLE}/Application Interface

This section discusses issues in, and problems arising from, interfacing \textit{RIPPLE} to external method application tools. A three tiered solution to this problem has been designed and implemented. Its structure and properties are discussed here. This structure allows both the mechanics of interfacing strategy and its semantic issues
and problems to be addressed. Chapter Six deals with more semantics used in the RIPPLE repository.

9.4.1 Interfacing Issues

The RIPPLE core system exists as a collection of routines using embedded QUEL in C [Ston76]. QUEL is the relational query language developed with the first Ingres system (now known as Academic Ingres) but is also supported in later commercial Ingres releases. Every RIPPLE model database contains a collection of core relations manipulated by these routines. In addition to the core relations are relations containing property information for items stored in the repository. The core routines provide a low level means for manipulating these relations. To use them effectively requires an intimate knowledge of both the structure of the core functions and relations. Additionally, many core relations extend through all the metadata layers maintained by RIPPLE. This further complicates the understanding of the core system internals by a person writing interface routines.

An external tool wishing to manipulate structures stored in the model database would have to call each appropriate RIPPLE function itself. This not only complicates the tool, but requires it to think in terms of RIPPLE constructs. This is contrary to the objective of allowing tools to communicate with RIPPLE on their own terms.

Clearly this approach is inadequate. What is required is some form of padding, or interface level, between the tool and RIPPLE. This interface could provide the translation from actions performed by the tool on its own constructs to the equivalent actions performed by RIPPLE core system functions.

9.4.2 A Three Tiered Approach

The diagram shown in figure 9.1 depicts how a tool could communicate with the interface layer. The application residing in the application layer calls functions provided in the interface layer described above. This layer in turn calls the appropriate RIPPLE functions residing in the core system. The results are then passed back to the interface layer and from there back to the tool. The application and core system layers communicate using the Sun Remote Procedure Call (RPC) and External Data Representation (XDR) libraries.

The interface layer would also enforce semantics appropriate to the operations it is performing. These semantic checks would be transparent to the design tool. They also allow the interface layer to reject operations and pass error conditions back to the application layer for actioning.
A closer inspection of the interface layer reveals a further structure shown in figure 9.2. Oval shaped objects in the two layers represent interface functions. In the upper layer of the interface reside routines that provide services dealing with design tool concepts. Actions such as delete an entity, delete a relationship, add an attribute to an entity etc... are examples of these.

Routines in the lower layer are called by the upper layer, never by the tool itself. Although these functions might be visible to the tool, it should only use those upper level interface functions provided for it. The lower level interface functions deal in terms of core system concepts (i.e. R/PPLE attributes, object etc...). Operations such as delete all members from an aggregation, delete a connection end, define an attribute and so on are examples of these and their interface requirements.

Dividing the interface layer in this manner allows the bottom level to remain a static collection of functions providing a diverse range of services. All of these services deal in terms of R/PPLE concepts and abilities and are fully supported in the core system. The top level may be a relatively dynamic set of functions, lending themselves to programmer modification and creation. These utilise services provided by the lower level of the interface. This structure provides a high degree
of extensibility and versatility – valuable attributes of an interface that must cater for, \textit{a priori}, unknown tools.

Routines at both interface layer levels may call other routines on the same level as well as routines ‘beneath’ them. For instance, \textit{delete an entity} may call the \textit{delete an attribute} function for each attribute present in the entity before finally deleting the entity itself. \textit{Delete an attribute} may in turn call \textit{delete a domain} to destroy the domain of the attribute it is to delete.

Some lower level interface functions may turn out to be highly generic across many tools. These may require little more than macro definitions within the upper level to make them appear specific to a particular tool.

Many of the routines in the lower half of the interface layer will be thinly disguised versions of actual core system routines. The chief advantage of this is the ability to easily package these as remote procedure calls. This enables \texttt{RIPPLE} to reside, compiled and statically linked, with the Ingres DBMS libraries and have calls to its routines made by the interface layer using remote procedure calls. Client programs and the \texttt{RIPPLE}-server may then communicate transparently via this interface.

There exist interface functions for over one hundred \texttt{RIPPLE} core system functions. Only those functions that make sense if provided for external use with have been included.

\subsection*{9.4.3 Semantic Issues}

The core system, on its own, enforces few semantics due to its simple, yet elegant, design philosophy. These semantics (described in detail in Appendix C) are sufficient to keep the \texttt{RIPPLE} core system in a constant state of integrity. They are not designed to be sufficient semantic controls for methods using \texttt{RIPPLE}. However, providing the building blocks from which more complex semantic controls may be effected is their aim.

For instance, the \texttt{RIPPLE} core system allows a connection to temporarily exist with one end, or no ends at all. This would not be acceptable if the connection were to represent a relationship in the Chen E-R Model and it is unlikely that a tool would use \texttt{RIPPLE} under these conditions. This section examines and discusses these restrictions.

The interface layer, described above, provides a useful place to enforce semantic constraints. The interface has two ‘layers’ – one which works with tool concepts, and the other with \texttt{RIPPLE} concepts.
The upper layer is used to enforce constraints applying to modelling constructs. The lower layer does little more than observe return codes from the core functions it calls, proceeding if the last operation succeeded, backing out and aborting if an error condition is detected. This approach is similar to the way transactions are handled in databases.

Defining a relationship in a Chen E-R modelling tool provides a simple example. According to the Chen E-R Model method definition, a relationship must connect one (if a self-relationship) or two (if a binary relationship) entities together. According to Ripple, a connection may exist with any number of ends permitted by the n-aryness domain defined for it in the method definition. Clearly an interface function that defines relationships needs to bridge this semantic gap to work correctly. This 'bridging' allows the model to remain in a state of integrity with respect to both the Chen ER modelling tool and Ripple. Figure 9.3 illustrates how this bridging takes place.

A blown-up view of the interface layer

The arrows (↔) serve to indicate that there exist more functions within the two interface layers.

Figure 9.3: A diagrammatic view of the layer interactions for creating a relationship in the Chen Entity Relationship Method

A tool in the application layer has issued a command to create a relationship between two entities. At this point the tool may apply any semantic constraints of its own.

It then calls the upper level interface function define relationship that accepts two or more entities and other information such as the relationship’s name and description plus any other properties. This function, along with the other upper level interface functions for this tool, have all been written by, for example, a programmer responsible for enabling that tool to utilise Ripple. At this point there
is another opportunity for semantic checks to be performed. It is, however, unlikely that they would be different from those performed by the tool before the interface function was called. It is easy to see that at least some of the burden for checking semantic correctness could be moved from the application tool to the upper interface layer. In this way the repository is supplying further services to the tool.

Define relationship then calls the lower level interface functions define connection and add part (to the connection), required to set up the connection. These functions then call (via the RPC interface) their respective RIPPLE functions (rpl_define_connection and rpl_add_connection_end). It is at the lower level of the interface that feedback from RIPPLE is encountered in the form of return codes.

If one of these functions detects and returns a fatal error condition then the interface must ensure that integrity is preserved with respect to the Chen E-R Modelling tool\(^1\). In this case it requires backing out of the define connection operation. This entails one of the following sets of actions:

(a) Return an error and do nothing else (case where connection define failed).

(b) Destroy connection and return an error (case where either of the connection end adds has failed). Note we do not distinguish which connection end add failed.

We simply call the connection destroy function which automatically takes care of removing all ends before destroying the connection.

Note that at the time the upper interface layer has received the fatal error condition the core system has already taken the necessary steps to ensure model integrity with respect to RIPPLE. This approach provides a high level of semantic integrity regards each tool's particular design information. In the case of the Chen E-R Model, a small set of top level interface routines consisting of the basic Create, Read, Update and Delete (CRUD) operations for domains, attributes, entities and relationships are required. Many of the lower level interface capabilities need not be echoed in the upper level, further hiding complexity.

If more powerful design methods are employed, more of the lower level interface may need to be exposed, and accessed via the upper level, to provide the functionality required by such tools.

9.5 An Example of Embedded QUEL

The following piece of code implements the function that adds an item to a connection. The basic routine in all RIPPLE functions performing many different

\(^1\) RIPPLE system integrity is maintained in all but the severest of errors (usually those arising from internal consistency check failures, DBMS or machine crashes).
functions is to check the validity of the passed arguments and then perform the operation. This is followed in the function listed below. The functions this function calls to perform these checking operations also used embedded QUEL to query the database for the information they require. The *rpl_* prefix (short for *ripple_*) indicates a *RIPPLE* function that performs a specific task. The *rc_* (short for *ripple_check_* ) prefix indicates functions that perform specific checks.

```c
/***************************************************************
Function Name: rpl_connection_add
Parameters: 1 - int: number of connection to add part to
            2 - int: number of part being added to connection
Returns: 3 - int: connection end concept class
Assumptions: None

This function adds an item to the definition of an already existing connection. Its type does not need to be specified.

****************************************************************/

#define int rpl_connection_add(ident, addition, end_concept)
#define ident, addition, end_concept;
#define num_items;

/* Check arguments */
if ( (rc validate db() == -1) ||
    (rc-connection exists(ident) == -1) ||
    (rc-connection-add(ident, addition, end_concept) == -1) ||
    (rc-concept(end_concept) == -1))
    return -1;

/* Check adding this item will not exceed item limit */
#define retrieve (num_items = count (connection_def.#part_id
    ##where #connection_def.#parent_conn = ident))
    if (rc_num_conn_items(num_items + 1) == -1)
        return -1;

/* get the identifier for this connection end */
rpl_nextid();

/* Append this new part to definition */
## append connection_def (#identifier = rpl_next_id,
    ## parent_conn = ident,
    ## part_id = addition)

/* Add this to the methods in use list */
if (rpl add to method(rpl_next_id, end_concept) == -1)
    return -1;

/* Record this in the histories of the connection and addition */
update_mh(ident, sprintf(msg, "%s added to connection",
    system_object_class_names[rpl_class_of(ident)]),
    addition);
update_mh(addition, "Added to connection", ident);
```
return errno?-1:TRUE;
}
Chapter Ten: Conclusions

10.1 Fundamental Points of Project – A Recap

*RIPPLE* was conceived as an intelligent metadata repository designed to support multiple design methods manipulating a model. *RIPPLE* exists as a core metadata management system in conjunction with a method definition system. Initially *RIPPLE* must be taught about the concepts and structures that a particular method will expect it to store and manipulate. Once *Method Definition* has been used to describe the structure of the information that *RIPPLE* is to store and maintain, the design database is set to begin storing model information.

The main aims and objectives of the *RIPPLE* project are:

- Be method independent. *RIPPLE* should not be yet another CASE tool supporting a fixed number of methods (Entity Relationship, Data Flow and Process Decomposition). Instead it should be capable of supporting any required methods.
- Provide information-sharing support between methods using *RIPPLE*.
- Be able to describe the repository requirements of a variety of methods, both physical and semantic, to *RIPPLE – Method Definition*.
- An integrated, unified, repository – i.e. no physical or artificial partitioning of the repository to make separate sections for separate tools.
- A fundamental repository conceptual model that is simple, yet contains the power and expressibility required to support arbitrary method requirements.
- Provide repository support to tools and systems without having to include the repository system within themselves. To this end provide an extensible interface separate from the core system, but which communicates with it.
- Demonstrate that the aims and objectives, once realised, fit together and perform in the way envisaged – demonstration of concept.

*RIPPLE*, proper, consists of a set of core functions and relations. Currently it is implemented using embedded QUEL in C and a commercial relational database platform (Ingres). An external interface using the Sun remote procedure call libraries enables *RIPPLE* to operate as a server executing requests from client tool applications (which may be running on other machines).

10.2 Achievements

This project has achieved all the aims and objectives set out in chapter 1:

- *RIPPLE* is independent of any particular method or method. There is no one method, or set of methods, supported by *RIPPLE* to the exclusion of
others. This feature is conspicuously absent in current CASE tools and other similar systems discussed in Chapter One.

- A conceptual model featuring simplicity and elegance of design yet provides the flexibility and power to underwrite the requirements of the RIPPLE repository system.
- Development of the Method Definition system used to describe methods to the RIPPLE repository system.
- Development of an external interface allowing systems to utilise the RIPPLE repository in a client/server arrangement. The interface is both flexible and extensible allowing any required tool interface to be incorporated simply and easily into it.
- Well supported mechanism for information sharing between methods.
- Mediation of tool interactions.
- Demonstration that the concepts and ideas embodied by all parts of the RIPPLE repository system work together in a useful and achievable fashion to attain the goals set for it.

10.3 Other Applications for RIPPLE

RIPPLE is far from just a system to provide repository support to tools implementing design methods. Its ability to define the structure of information it is to store allows it to fulfil many other roles. In a broad sense, RIPPLE acts a database management system. The structures of tables required for the database are described to Method Definition and the core system manages data stored in them. This is, however, no claim to fame. Database systems are capable of doing this, and more, with far greater efficiency. However, the facilities provided by Method Definition and the flexibility of the application interface lend themselves to other tasks, some of which are briefly mentioned in the remainder of the section.

10.3.1 Method/Application Prototyping

RIPPLE could provide excellent support to someone working on a new or variant design method (also called meta-modelling). Using RIPPLE, a tool written to implement such a method need not concern itself with managing the database itself, but could use RIPPLE to provide those services. Later, when finalised, the tool could be extended to maintain its own design database. RIPPLE can also provide semantic support and a platform for evaluating the behaviour of the new method with other methods.
In a slightly more generic example, Ripple could be used in a similar fashion to lend support to an application being prototyped. In prototyping an application requiring database support, valuable time must be used to provide that database support to it. If Ripple was used, the structures required could be described to Method Definition. The interface code would be quite simple and quick to produce. The end result is cheap and quick to produce database support for an application prototype.

10.3.2 Impact Reporting & Repository Support for DBMSs
Databases are now in almost ubiquitous use. From airline reservations to supermarket stock inventories databases are used to store and maintain vast volumes of information. More often than not, such databases do not have a static structure. The existence of large numbers of Database Administrators (DBA) jobs depend on it! However, modifying databases can be a tricky business. Deleting a key attribute can have disastrous consequences – the least of which is the probable loss of the DBA’s job. Further, in the absence of knowledge about the structure of that information, those consequences may not be apparent until the deed is done. As the saying goes, shutting the gate after the horse has bolted is too late.

Ripple could provide a useful service by maintaining this information structure – metadata! Proposed changes to the structure of the database could be made first on the model maintained by Ripple. This allows the DBA to ascertain the changes that will take place by examining areas that have been effected by the change. Alternatively it could be the system providing repository support to a database management system that would, itself, provide these facilities to the DBA.

10.3.3 Schema Integration and Data Exchange
A methodology for data schema integration in the Entity Relationship model is described in [Bati83]. By storing the descriptions of the schemas being integrated Ripple could provide support to schema integration tasks.

In a similar fashion, Ripple could provide support to applications converting data (such as information stored in GIS systems) between different formats. Again Ripple could store descriptions of the formats and via the information sharing and concept mapping mechanisms aid the process of conversion.

10.4 Possible Improvements and Extensions
Whilst this thesis has demonstrated the intrinsic benefits and advantages of the approaches taken by Ripple, there are a number of improvements and extensions that could be applied to it. Due to the constraints and time limits imposed by a
thesis, not everything that would be nice to have in Ripple could be done. Extensive and complete interface specifications and their implementations for a variety of other useful methods may be implemented. Actual connections to real tools could be set up. Numerous smaller changes such as support for extra abstraction mechanisms could be added to the Ripple core system. While such mechanisms would, in fact, be easily supported by the tools in question through Ripple it is a more elegant solution if such support is supplied directly.

The following list describes a number of the larger improvements and extensions that could be made to Ripple. In line with the Ripple's stated purpose of being a research and teaching vehicle, some of these improvements would suit being the subject of Honours or Stage 3 projects.

- More complete interfaces for methods interfacing with Ripple as well as interfaces for actual design tools.
- Greater semantic control. An attractive method is to create semantic rules in a simple language that manipulates Ripple core constructs. These rules would then be interpreted and applied at run-time.
- Building more intelligence in the information sharing mechanisms. For instance, the problem mentioned in section 7.2.4 regarding importing connections could be researched to allow more freedom in sharing connection based information.
- A more comprehensive version management system for items stored in the repository. This extended version management system would allow, among other abilities, sets of items with particular versions to be created. Such a set could specify the state or a snapshot of the model at a given point in time. This allows the development state of the model to be recorded for releases of software systems used for testing.

10.5 Concluding Remarks

In the course of developing Ripple a number of aims and objectives were identified in Chapter One as being important in the design such a system. A layered abstraction was developed to allow the expression of concepts in a wide range of methods. From this abstraction, and the examination of a number of existing methods, the design of the repository conceptual model was distilled. Once formed, the conceptual model drove the design and construction of the Ripple core repository system. In conjunction with the conceptual model this core system provided the basis for the development of a method definition system allowing new methods to be described to the repository. This was all topped off with the construction of an extensible interface to allow other applications to use the
repository services provided by RIPPLE. This interface allows RIPPLE to behave as a repository server to a number of client tool applications. Interfaces may be rapidly developed for an application removing the need for it to provide such a system by itself.

At this point Toolset was designed and built to demonstrate the feasibility and workability of RIPPLE. A number of methods were described to the system and a simple model was created and manipulated. This system successfully demonstrated the aims and objectives stated at the beginning of this thesis have been achieved.

It is hoped that the experience gained in the design and construction of RIPPLE can provide more than an academic curiosity. Flaws present in integrated CASE toolsets, and other similar software design systems, have been redressed in RIPPLE. Solutions to these problems, and the ways they were approached and overcome may provide insights into improving other software and database design systems. The implementation of RIPPLE in a portable fashion allows other researchers to use this system.

In conclusion, RIPPLE has proved to be a successful exercise in repository design. Its power and flexibility provides a high degree of generic repository support for tool applications requiring it.
References


[Camp87]  Campbell, I. Standardization, availability and use of PCTE. Information and Software Technology 29, 8 (October 1987), 411-414.


[Chen76]  Chen, P.P.S. The Entity Relationship Model - Toward a Unified View of Data.. ACM Transactions on Database Systems 1, 1 (March 1976), 9-36.


Information Engineering Workbench, KnowledgeWare Inc, 3340 Peachtree Road, N.E., Atlanta GA, USA.


[Ston76] Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation of INGRES. ACM Transactions on Database Systems 1, 3 (September 1976), 159-222.


Appendix A : Glossary Of Terms

This appendix describes the terms and concepts used within this thesis.

In “information engineering” there are many common terms and concepts. Some have different names and meanings given them by different researchers. Depending on the context or researcher using them, some mean entirely different things. Such overloading and impreciseness of terms can make it difficult to establish a common terminology.

Note: The source of a terms’ definition is included with the definition itself where more than one exists for it. Typically the terms and definitions defined by [Teor86] and [Date86b] are used.

- **Attribute**: A property capable of storing a single atomic piece of information. This is very similar to the relational idea of an attribute.

- **Connection**: A set of items ‘connected’ together. Connections describe relationships, or connectivity, between items. They are used to describe, for example, relationships and Data Flow Diagram data flows.

- **Connection End**: A connection has a number of ends. Each end attaches to some item, including it in the definition for that connection.

- **Data**: The Oxford Reference Dictionary defines data as “Facts or information used as a basis for inference or reckoning or prepared for being processed by a computer”. For instance, supplier “Smith” supplies the part “10cm right threaded stainless steel bolt”.

- **Domain**: “... a pool of values, from which one or more attributes draw their values.” [Date86b] (pg. 235). This is very similar to the relational idea of a domain. **RIPPLE** only caters for simple domains. Composite domains are implicitly created by aggregating attributes (not domains) into objects.

- **Domain Part**: A sub-structure making part of a domain’s definition. Three types of domain parts exist – simple, enumerated and abstract. Simple domain parts define ranges of values. Enumerated domain parts explicitly enumerate sets of values. Abstract domain parts are actually domains proper but which are used as part of another domains definition.
**Identifier**

Every item, including methods, stored by Ripple is assigned a system wide unique identifier. This identifier is unique and is shared by no other item in a model database.

**Metadata**

Data about the structure of, that describes, or characterises data. Eg. “Suppliers supply Parts”.

**Meta-metadata**

Data about the structure of metadata. Eg. “Chen Entity Relationship relationships may have (1:1), (1:N), & (M:N) cardinalities”.

**Meta-meta-metadata**

Data about the structure of meta-metadata. Eg. “Entity Relationship and Data Flow methods have common information: E-R entities may be used as data flowing along a data flow in a Data Flow Diagram”.

**Method**

A generic term for the method or application used to manipulate or view model information stored within a Ripple model database. Entity Relationship, Data Flow and dependency analysis tools implement examples of what Ripple considers ‘methods’.

**Method Definition**

The description of a method in terms of the concepts and constructs it manipulates. Attributes, entities and relationships are concepts (of the Chen E-R Model among others). Abstraction of attributes is the way the methods build entities. A method definition also describes the manner in which other methods may import items created by this method.

**Model**

The description of a problem or design stored by Ripple. It may be a set of Entity Relationship diagrams, Data Flow diagrams, process decomposition diagrams etc... All of these may exists at the same time in the same Ripple model database.

**Modification History**

A set of entries describing the evolution over time (from creation until destruction) of an item stored by Ripple in a model database.

**Object**

A set of items abstracted together via an abstraction mechanism. Entity Relationship entities are stored as a Ripple objects. Aggregation, collection, generalisation and subset hierarchy abstraction mechanisms may be used.
Object Class

An object class is used to denote attribute, domain, domain part, object and connection classes of items stored in a model database. Methods are also considered to be an object class by RIPPLE. A further class of items is also recognised – connection ends. This is, however, completely transparent to methods using the system.

Relation

A relation as used by relational databases. RIPPLE uses the relational Ingres database system to store information. Model information is stored in relations in an Ingres database.

Repository

The Oxford Reference Dictionary defines a repository as “Receptacle, place where things are stored or may be found.” In the information engineering context, a repository is a place where design information may be stored, retrieved from and manipulated by design tools.

Semantics

“... the study of the relationships between signs and symbols and what they represent...” – Collins concise English Dictionary. The rules governing how and when operations may be performed upon objects being manipulated. See also semanticist - one who studies semantics.

Unlink (of an item)

When a method wishes to discontinue using an item it removes it from its inuse list. The item is not deleted, it is simply no longer recorded as being used by that method. Thus the item has been unlinked from it. The method may, in the future, re-add this item to its inuse list if it still exists in the repository.

The following are definitions of the abstraction mechanisms used by RIPPLE to create objects from other items. These abstraction principles are used in the sense of Teorey, Yang and Fry [Teor86].

Aggregation

Aggregation is the construction of an object by the explicit combination of a number of other items.

Generalisation

Generalisation of a class of objects in order to hide the individual differences between them. For instance generalising the set {Dogs, Cats, Elephants} to Animals.

† A detailed explanation and examples of these types of abstraction may be found in section 3.6.2.
Note: The sets being generalised are mutually exclusive. This is a consequence of using an attribute to differentiate between the generalised items.

**Subset Hierarchy**

Similar to a generalisation but deals with items that are subsets of some other set. For instance the set people form a super set of the subsets {Male People, Female People, Student People, Worker People, Retired People...}.

Note: These subsets are not mutually exclusive. They are analogous to Pascal (tagged) variant records.
Appendix B: *RIPPLE* Core System Relation Structure

B.1 Introduction

This first section of this appendix contains a complete set of Extended Entity Relationship diagrams for the *RIPPLE* core system. The second section of this appendix describes the relations used to store information in the *RIPPLE* core system. Each relation is described by a table, the structure of which is as follows:

Each table has three columns. The first states the name of the attribute. If this is italicised (e.g. *Identifier*) then the attribute is a primary key. The second column contains the QUEL type of this attribute. The letter indicates the type (*i* = integer, *c* = character string, *f* = floating point and *date* = date field). The number indicates the attributes' size. For integers and character strings this is the number of bytes used to represent it. Floating point types are either 4 or 8 byte representations while QUEL dates have a fixed format.

B.2 Extended Entity Relationship Representation

B.2.1 Introduction

This section provides a diagrammatic representation of the core relation structures in *RIPPLE*. Extended Entity Relationship (EER) diagrams have been used to provide this representation.

Presenting the entire structure as a single diagram has proven very difficult. To facilitate reader understanding this structure has been split into a number of smaller diagrams. Descriptions of entities used within this section may be found in Appendix D.

B.2.2 Attributes & Domains

Attributes have a *domain* from which they draw their values. These in turn consist of domain parts (*simple, enumerated or abstracted*). Simple domain parts may have constraints placed on them such as "value less than or equal to 19". Enumerated domain parts consist of a set of values. This set of values either defines the values within the enumeration or those not within it. An abstracted domain part is, in fact, just another domain. These allow complex domain definition hierarchies to be built (an examples of which may be found in section 3.6.1.2). Figure B.2.1 describes their structure in *RIPPLE*.
B.2.3 Objects

Objects are created by applying an abstraction mechanism to a set of items stored in the repository. A more detailed explanation of how objects are created may be found in section 3.6.2. The set of concepts that may be involved in a particular abstraction are specified in *cm_object*. The structure of the object sub-system is shown in figure B.2.2.

B.2.4 Names

All items stored by *RIPPLE* have an identifier, name and a description filed with them. Names are filed in a separate relation, called *names*. This relation is used to make life easier in the *RIPPLE* core machinery. Although the name and description may be changed, an identifier stays with the item for its entire lifetime and may not be changed. The relationship of names to nameable items in *RIPPLE* is shown in figure B.2.3.
B.2.5 Connections

Connections may connect any set of items stored by \textit{RJPPLE}. The types of items that may be connected by a particular connection concept are specified in the method definition (discussed in detail in Chapter Five). A connection consists of the connection per se, and a description of the items participating in the connection (see figure B.2.4).

![Connection Diagram](image)

**Figure B.2.4:** EER representation of connection structure in \textit{RJPPLE}.

B.2.6 Which method uses which items?

An item may be imported, and used in other methods in the repository. The \textit{method ids} entity, shown in figure B.2.5, is used to record which methods are using which items.
B.2.7 Information Sharing

Methods can grant permissions to other methods to import instances of concepts that they manipulate. Methods both grant, and receive, these permissions.

B.2.8 Modification Histories

Every item stored by RIPPLE has a modification history recorded for it. This history tracks all changes made to an item. Figure B.2.7 shows its relationship with other entities in RIPPLE. Though not shown in Figure B.2.7, all relationships between the Modification History entity and the other entities in the diagram are mutually exclusive.
B.2.9 Method Definition Structures

These structures are responsible for storing meta-metadata – the descriptions of methods being used to manipulate the design model.

The attribute, domain, object and connection entities are used in most of the diagrams shown above. These all have a (1 : 1) relationship with the concept entity shown in figure B.2.8. It is through this relationship that constraints are placed on what may be done with particular concepts.

For instance, assume there exist two concepts (A & B) both based on the \textit{RIPPLE object} concept. We may state that an instance of concept A may be used to construct an instance of concept B, but that we may not use an instance of concept B to construct an instance also of concept B.

This is not obvious from many of these diagrams as they deal with \textit{RIPPLE} base concepts. It becomes apparent when the structure of figure B.2.7 and the entity structures described in Appendix B.3 are studied.
B.3 Entity Structures & Descriptions

The following entity descriptions contain comma separated entries comprising their attributes. Those depicted in italics are key attributes. Secondary key attributes are underlined. All primary and foreign keys are RFPPE identifiers of other items stored in the system.

<table>
<thead>
<tr>
<th>Entity Type</th>
<th>Attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method_IDs</td>
<td>(Method, Identifier, Class, Concept_ID)</td>
</tr>
<tr>
<td>Names</td>
<td>(Name, Type, Identifier, Based_On)</td>
</tr>
<tr>
<td>Attribute</td>
<td>(Identifier, Name, Description, Domain, Flags, MetaClass)</td>
</tr>
<tr>
<td>Domain</td>
<td>(Identifier, Name, Description, Operator, Consistent, MetaClass)</td>
</tr>
<tr>
<td>Domain Part</td>
<td>(Identifier, Part_ID, Part_Type)</td>
</tr>
<tr>
<td>Simple Domain Part</td>
<td>(Identifier, Name, Description, Base_Type, Base_Type_Size, MetaClass)</td>
</tr>
<tr>
<td>Domain Constraint</td>
<td>(Identifier, Type, Value)</td>
</tr>
<tr>
<td>Enumerated Domain Part</td>
<td>(Identifier, Name, Description, Base_Type, Base_Type_Size, Constraint_Type, MetaClass)</td>
</tr>
<tr>
<td>Enumeration Value</td>
<td>(Identifier, Part, Value)</td>
</tr>
<tr>
<td>Object</td>
<td>(Identifier, Name, Description, Abstraction_Type, Flags, Reference, MetaClass)</td>
</tr>
<tr>
<td>Object Part</td>
<td>(Identifier, Part_Type, Part_ID)</td>
</tr>
<tr>
<td>Connection</td>
<td>(Identifier, Name, Description, MetaClass)</td>
</tr>
<tr>
<td>Connection End</td>
<td>(Identifier, Part, Part_ID, Concept)</td>
</tr>
<tr>
<td>Modification History</td>
<td>(Identifier, Number, Mod_By, Mod_Date, Description, Reference)</td>
</tr>
<tr>
<td>Method</td>
<td>(Identifier, Name, Description, Creator_Name, Created_When, Last_Modified, InUse)</td>
</tr>
<tr>
<td>Concept</td>
<td>(Identifier, Name, Description, Base_Concept, POID)</td>
</tr>
<tr>
<td>cm_object</td>
<td>(Identifier, Ab_types)</td>
</tr>
<tr>
<td>cm_object_construct</td>
<td>(Identifier, Ab_type, Concepts)</td>
</tr>
<tr>
<td>cm_connection</td>
<td>(Identifier, degree)</td>
</tr>
<tr>
<td>cm_conn_conn_end</td>
<td>(Conn_Concept, End_Concept)</td>
</tr>
<tr>
<td>cm_conn_end</td>
<td>(Identifier, End_Concepts)</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>----------------------------</td>
</tr>
<tr>
<td>Import_permission</td>
<td>(Method_from, Concept_from, Method_to, Concept_to)</td>
</tr>
</tbody>
</table>
Appendix C: Concept Demonstration Methods

C.1 Introduction

This appendix contains a number of screen snaps from the Method Definition system that show all the items present in each of the methods manipulating the demonstration model. They are included here for completeness and to improve clarity in Chapter Eight.

The definitions of the three methods in question are by no means complete. In some places extra import permissions, concept properties and the like could have been included. However, as they serve only to demonstrate the capabilities of the RIPPLE Method Definition system and Toolset tool applications completeness is not necessary.

C.2 Chen E-R Method Method Definition Components

Figure C.1 shows all the domains within in the Chen E-R method definition. The domains cm_domain* are domains created by RIPPLE during particular parts of the method definition. The first is used to describe which method concepts may take part in the construction of a Chen E-R entity (i.e. attributes). The second is used to describe how ends of Chen E-R relationships may connect to instances of other method concepts (i.e. entities).

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Op</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>CER Entity Restriction</td>
<td>CER RelEnd Description</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>CER RelEnd Description</td>
<td>CER RelEnd Descriptions are 250 character</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>Cardinality values domain</td>
<td>Holds the allowed degree values for relations</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>Key type values</td>
<td>Cardinality value for the connection between</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>cm_domain29</td>
<td>Values for key type of attributes. Types</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>cm_domain33</td>
<td>cm_object_constrct abstractable concepts</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>cm_conn_end attachable concepts holder (dom)</td>
<td>cm_conn_end attachable concepts holder (dom)</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Figure C.1: Chen E-R method definition meta-metadata domains.
Figure C.2 shows all the attributes created in the Chen E-R method definition. The *End cardinality attribute* and *CER description from end* attributes could also be shared by the Extended E-R method.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Domain Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>CER Attribute</td>
<td>CER Description from end</td>
</tr>
<tr>
<td>End Cardinality</td>
<td>End Cardinality values domain</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4)

Figure C.2: Chen E-R method definition meta-metadata attributes.

Figure C.3 shows all objects created in the Chen E-R method definition. The *no properties* object is shared with the Extended E-R and Data Flow models. This object is used as the *Property Object* for all concepts not having properties. Objects whose names end in *POID* are used as property objects for the method concept referred to in their description.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CER Attribute POID</td>
<td>CER Relationship End POID</td>
</tr>
<tr>
<td>CER Relationship</td>
<td>List of attributes in the entity at this end</td>
</tr>
<tr>
<td>Foreign</td>
<td>Vanilla object for concepts with no properties</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4)

Figure C.3: Chen E-R method definition meta-metadata objects.
Figure C.4 shows all concepts created in the Chen E-R method definition.

<table>
<thead>
<tr>
<th>Concept Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen ERM attribute</td>
<td>Describes Chen ERM attribute concept</td>
</tr>
<tr>
<td>Chen ERM domain</td>
<td>Describes Chen ERM domain concept</td>
</tr>
<tr>
<td>Chen ERM entity</td>
<td>Describes Chen ERM entity concept</td>
</tr>
<tr>
<td>Chen ERM relationship</td>
<td>Describes Chen ERM relationship concept</td>
</tr>
</tbody>
</table>

Figure C.4: Chen E-R method definition concepts.

Figure C.5 shows all import permissions created in the Chen E-R method definition. These permissions cover concepts in both the Extended E-R and Data Flow methods.

<table>
<thead>
<tr>
<th>Originating Concept</th>
<th>Importing Method</th>
<th>New Local Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chen ERM attribute</td>
<td>Extended Entity Relations</td>
<td>Extended ERM attribute</td>
</tr>
<tr>
<td>Chen ERM entity</td>
<td>Extended Entity Relations</td>
<td>Extended ERM entity</td>
</tr>
<tr>
<td>Chen ERM relationship</td>
<td>Extended Entity Relations</td>
<td>Extended ERM relationship</td>
</tr>
<tr>
<td>Chen ERM relationship end</td>
<td>Data Flow Diagram</td>
<td>IPO attribute</td>
</tr>
</tbody>
</table>

Figure C.5: Chen E-R method definition import permissions.
C.3 Extended E-R Method Method Definition Components

Figure C.6 shows all attributes created in the Extended E-R method definition. As with the Chen E-R method, the cm_domain* domains relate to object construction and connection specification for relationships. Domains Cardinality values domain and Key type values have been imported from the Chen E-R method definition.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Op</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality values domain</td>
<td>Cardinality value for the connection between</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>EER Membership Class</td>
<td>Membership classes for connections between</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>EER ReEnd Description</td>
<td>EER ReEnd Description are 250 character</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>EER relationship degree</td>
<td>Holds the allowed degree values</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>Extended Entity Abstract</td>
<td>Holds the allowed abstraction types for Ext</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>Key type values</td>
<td>Values for key type of attributes. Types of</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>cm_domain101</td>
<td>cm_object_eonstruct_abstractable</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>cm_domain105</td>
<td>cm_object_eonstruct_abstractable</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>cm_domain109</td>
<td>cm_object_eonstruct_abstractable</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>cm_domain111</td>
<td>cm_object_eonstruct_abstractable</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>cm_domain97</td>
<td>cm_object_eonstruct_abstractable</td>
<td>Yes</td>
<td>-----</td>
</tr>
<tr>
<td>cm_domain99</td>
<td>cm_object_eonstruct_abstractable</td>
<td>Yes</td>
<td>-----</td>
</tr>
</tbody>
</table>

Figure C.6: Extended E-R method definition meta-metadata domains.

Figure C.7 shows all attributes created in the Extended E-R method definition. Note the use of the Cardinality values domain imported from the Chen E-R method.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Domain Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>EER Attribute</td>
<td>EER Attribute Keywords</td>
</tr>
<tr>
<td>EER Description from end</td>
<td>EER ReEnd Description</td>
</tr>
<tr>
<td>EER End Cardinality</td>
<td>EER Membership Class</td>
</tr>
<tr>
<td>EER Membership Class</td>
<td>Cardinality values domain</td>
</tr>
</tbody>
</table>

Figure C.7: Extended E-R method definition meta-metadata attributes.
Figure C.8 shows all objects created in the Extended E-R method definition. The *No properties* object is shared by all methods and is used as a *Property Object* for concepts with no properties. Objects whose names end in *POID* are used as property objects for the method concept referred to in their description.

![Table of Objects in Method Definition](image)

Figure C.8: Extended E-R method definition meta-metadata objects.

Figure C.9 shows all concepts created in the Extended E-R method definition. The *Extended ERM CER Entity* and *Extended ERM CER Relationship* concepts are used as mirror concepts to allow Chen E-R relationships to be imported into the Extended E-R method without violating constraints such as those regarding relationship degree (discussed in section 7.2.5).

![Table of Concepts in This Method](image)

Figure C.9: Extended E-R method definition meta-metadata concepts.
Figure C.10 shows all import permissions created in the Extended E-R method definition. These allow the Chen E-R and Data Flow methods to share its attributes. Other permissions which could have been created would also allow Extended E-R domains to be shared by the two other methods.

<table>
<thead>
<tr>
<th>Method Import Permissions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Current Method: Extended Entity Relationship</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Permissions List</th>
</tr>
</thead>
<tbody>
<tr>
<td>Originating Concept</td>
</tr>
<tr>
<td>Extended E-R attribute</td>
</tr>
<tr>
<td>Extended E-R attribute</td>
</tr>
</tbody>
</table>

Create(1) Delete(2) Find(3) Help(4) End(5)

Figure C.10: Extended E-R method definition meta-metadata import permissions.

C.4 Data Flow Method Method Definition Components

Figure C.11 shows all domains created in the Data Flow method definition. As with the two Entity Relationship methods, the cm_domain* domains relate to object construction and connection specification for data flows.

<table>
<thead>
<tr>
<th>Domains in Method Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>DFD Abstraction Types**</td>
</tr>
<tr>
<td>DFD Flow End Description</td>
</tr>
<tr>
<td>DFD Process Number</td>
</tr>
<tr>
<td>DFD external entity type</td>
</tr>
<tr>
<td>DFD source process number</td>
</tr>
<tr>
<td>cm_domain64</td>
</tr>
<tr>
<td>cm_domain66</td>
</tr>
<tr>
<td>cm_domain68</td>
</tr>
<tr>
<td>cm_domain70</td>
</tr>
<tr>
<td>cm_domain74</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(6)

Figure C.11: Data Flow method definition meta-metadata domains.
All attributes created in the Data Flow method definition.

Note: The description of DFD destination process should read “DFD destination process number”.

<table>
<thead>
<tr>
<th>Attribute Name</th>
<th>Domain Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFD External Entity Type</td>
<td>DFD external entity type</td>
</tr>
<tr>
<td>DFD Process Number</td>
<td>DFD Process Number</td>
</tr>
<tr>
<td>DFD destination process</td>
<td>DFD source process number</td>
</tr>
<tr>
<td>DFD source process</td>
<td></td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4)

Figure C.12: Data Flow method definition meta-metadata attributes.

Figure C.13 shows all objects created in the Data Flow method definition. Objects whose names end in POID are used as property objects for the method concept referred to in their description.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFD Data Flow End POID</td>
<td>DFD Data Flow POID</td>
</tr>
<tr>
<td>DFD Data Flow POID</td>
<td>DFD Data Flow POID</td>
</tr>
<tr>
<td>DFD External Entity POID</td>
<td>DFD External Entity POID</td>
</tr>
<tr>
<td>DFD Process POID</td>
<td>DFD Process POID</td>
</tr>
<tr>
<td>DFD flowing data</td>
<td>Holds all the data flowing along a data flow</td>
</tr>
<tr>
<td>DFD subprocesses</td>
<td>Holds all the processes that this process extends</td>
</tr>
<tr>
<td>No Properties</td>
<td>Vanilla object for concepts with no properties</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4)

Figure C.13: Data Flow method definition meta-metadata objects.
Figure C.14 shows all concepts created in the Data Flow method definition. The DFD data flow data concept is used in a similar manner as an E-R entity. It allows groupings of data moving along a flow to be constructed. Many such groupings may be stored as a data flows' properties.

<table>
<thead>
<tr>
<th>Concept Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFD attribute</td>
<td>Describes DFD attribute concept</td>
</tr>
<tr>
<td>DFD data flow</td>
<td>Describes DFD data flow concept</td>
</tr>
<tr>
<td>DFD data flow data</td>
<td>Describes DFD data flow data concept</td>
</tr>
<tr>
<td>DFD domain</td>
<td>Describes DFD domain concept</td>
</tr>
<tr>
<td>DFD external entity</td>
<td>Describes DFD external entity concept</td>
</tr>
<tr>
<td>DFD file store</td>
<td>Describes DFD file store concept</td>
</tr>
<tr>
<td>DFD process</td>
<td>Describes DFD process concept</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4)

Figure C.14: Data Flow method definition meta-metadata domains.

Figure C.15 shows all import permissions created in the Data Flow method definition. Other permissions that could be added would allow the two Entity Relationship methods to import domains created in the Data Flow method.

<table>
<thead>
<tr>
<th>Originating Concept</th>
<th>Importing Method</th>
<th>New Local Concept</th>
</tr>
</thead>
<tbody>
<tr>
<td>DFD attribute</td>
<td>Their Entity/Relationship</td>
<td>Their ERM attribute</td>
</tr>
<tr>
<td>DFD attribute</td>
<td>Extended Entity Relations</td>
<td>Extended ERM attribute</td>
</tr>
</tbody>
</table>

Create(1) Delete(2) Find(3) Help(PF4) End(PF3)

Figure C.15: Data Flow method definition meta-metadata import permissions.
Appendix D: Concept Demonstration Model

D.1 Introduction

This appendix contains a number of screen snaps from the Toolset application showing view seen by each of the methods manipulating the demonstration model. They are included here for completeness and to improve clarity in Chapter Eight.

The demonstration model presented in Chapter Eight is by no means a detailed one. Obviously the fragments shown would exist as part of a larger whole—perhaps an information system operated by an engineering parts supplier to keep track of their business. However, this is intended only as a simple demonstration model and has been kept correspondingly small and simple.

D.2 Chen E-R View of Demonstration Model

Figure D.1 shows all domains present in the Chen E-R method. While all of the attributes that use these domains have been been imported into the Extended E-R method and some have been imported into the Data Flow method, none of the domains here has been imported into either of the other methods.
Figure D.2 shows all the attributes created in the Chen E-R method. A large number of these attributes have been imported by the Extended E-R and Data Flow methods.

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Category/contype</td>
<td>Category/contypes</td>
</tr>
<tr>
<td>Employee Age</td>
<td>Employee ages</td>
</tr>
<tr>
<td>Employee Name</td>
<td>Employee names</td>
</tr>
<tr>
<td>Employee Number</td>
<td>Employee numbers</td>
</tr>
<tr>
<td>Employee Type</td>
<td>Employee types</td>
</tr>
<tr>
<td>Employee Union</td>
<td>Union names</td>
</tr>
<tr>
<td>Part Description</td>
<td>Part descriptions</td>
</tr>
<tr>
<td>Part Name</td>
<td>Part names</td>
</tr>
<tr>
<td>Part Number</td>
<td>Part numbers</td>
</tr>
<tr>
<td>Part Quantity</td>
<td>Part quantities</td>
</tr>
<tr>
<td>Part cost</td>
<td>Part costs</td>
</tr>
<tr>
<td>Salary</td>
<td>Employee salaries</td>
</tr>
<tr>
<td>Shipment Number</td>
<td>Shipment numbers</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) View Properties(4) Delete(5) >

Figure D.2: All attributes in the Chen E-R method.

Figure D.3 shows all the entities created in the Chen E-R model. All of these entities were imported into the Extended E-R model while the Data Flow model created its own data flow data objects from attributes it imported from this method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee Manager</td>
<td>Manager</td>
</tr>
<tr>
<td>Part</td>
<td>Parts entity</td>
</tr>
<tr>
<td>Shipment</td>
<td>Shipment of parts</td>
</tr>
<tr>
<td>Supplier</td>
<td>Supplier of parts</td>
</tr>
<tr>
<td>Worker</td>
<td>Worker</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4) >

Figure D.3: All entities in the Chen E-R method.
Figure D.4 shows all the relationships created in the Chen E-R model. All of these relationships, bar Employee/Manager and Employee/Worker are imported by the Extended E-R method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee/Manap</td>
<td>In employee may be a manager</td>
</tr>
<tr>
<td>Employee/Worker</td>
<td>An employee may be a worker</td>
</tr>
<tr>
<td>Shipment/Port</td>
<td>Shipment contains part</td>
</tr>
<tr>
<td>Supplier/Employee</td>
<td>Supplier employs employees</td>
</tr>
<tr>
<td>Supplier/shipment</td>
<td>Supplier ships shipment</td>
</tr>
</tbody>
</table>

Figure D.4: All relationships in the Chen E-R method.

D.3 Extended E-R View of Demonstration Model

Figure D.5 shows all the domains in the Extended E-R method. As can be seen this list is empty! The reason for this is that the Extended E-R method has imported all its attributes from the Chen E-R method. It has not, however, imported the attributes’ domains, nor have these domains been imported as a consequence of importing the attributes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Op</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure D.5: All domains in the Extended E-R method.
Figure D.6 shows all the attributes in the Extended E-R method. All of these attributes have been imported from the Chen E-R method.

![Extended ERM attributes in model](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee Age</td>
<td>Employee Ages</td>
</tr>
<tr>
<td>Employee Name</td>
<td>Employee names</td>
</tr>
<tr>
<td>Employee Number</td>
<td>Employee numbers</td>
</tr>
<tr>
<td>Employee Type</td>
<td>Employee types</td>
</tr>
<tr>
<td>Employee Union</td>
<td>Union names</td>
</tr>
<tr>
<td>Part Description</td>
<td>Part descriptions</td>
</tr>
<tr>
<td>Part Name</td>
<td>Part names</td>
</tr>
<tr>
<td>Part Number</td>
<td>Part numbers</td>
</tr>
<tr>
<td>Part cost</td>
<td>Part costs</td>
</tr>
<tr>
<td>Salary</td>
<td>Employee salaries</td>
</tr>
<tr>
<td>Shipment Number</td>
<td>Shipment numbers</td>
</tr>
<tr>
<td>Supplier Name</td>
<td>Supplier names</td>
</tr>
</tbody>
</table>

Figure D.6: All attributes in the Extended E-R method.

Figure D.7 shows all the entities created in the Extended E-R method. Only one is shown as this method has imported the other entities it needs from the Chen E-R method. These are shown in figure D.8.

![Extended ERM entities in model](image)

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Employee -- Worker</td>
<td>Employee Generalization</td>
</tr>
</tbody>
</table>

Figure D.7: All entities in the Extended E-R method.
Figure D.8 shows all the entities imported by the Extended E-R method from the Chen E-R method.

Figure D.8: All entities in the Extended E-R method imported from the Chen E-R method.

Figure D.9 shows all the relationships created in the Extended E-R method. None is present as this method has imported all the relationships it needs from the Chen E-R method (shown in figure D.10).

Figure D.9: All relationships in the Extended E-R method.
Figure D.10 shows all the relationships imported by the Extended E-R method from the Chen E-R method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shipment/Part</td>
<td>Shipment contains part</td>
</tr>
<tr>
<td>Supplier/Employee</td>
<td>Supplier ships shipment</td>
</tr>
<tr>
<td>Supplier/shipment</td>
<td></td>
</tr>
</tbody>
</table>

Figure D.10: All relationships in the Extended E-R method imported from the Chen E-R method.

D.4 Data Flow View of Demonstration Model

Figure D.11 shows all domains created in the Data Flow method. Domains of attributes imported from the Chen E-R method are not shown as they are not imported as a consequence of importing these attributes.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Status Codes</td>
<td>Status codes returned from create new part</td>
</tr>
<tr>
<td>Status messages</td>
<td>Status messages returned by report status</td>
</tr>
</tbody>
</table>

Figure D.11: All domains in the Data Flow method.
Figure D.12 shows all attributes present in the Data Flow method. All the Part attributes have been imported from the Chen E-R model. The Status attributes have been created in the Data Flow method.

![DFD attributes in model](image)

Figure D.12: All attributes in the Data Flow method.

Figure D.13 shows all the data flow objects in the Data Flow method. These have all been created in this method.

![DFD data flow objects in model](image)

Figure D.13: All data flow objects in the Data Flow method.
Figure D.13 shows all the processes in the Data Flow method.

<table>
<thead>
<tr>
<th>DF D process in model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Assign part number</td>
</tr>
<tr>
<td>Create new part</td>
</tr>
<tr>
<td>Report status</td>
</tr>
<tr>
<td>Validate Part Details</td>
</tr>
</tbody>
</table>

Figure D.14: All processes in the Data Flow method.

Figure D.13 shows all the external entities in the Data Flow method.

<table>
<thead>
<tr>
<th>DFD external entities in model.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>Clerk</td>
</tr>
</tbody>
</table>

Figure D.15: All external entities in the Data Flow method.
Figure D.13 shows all the file stores in the Data Flow method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parts File</strong></td>
<td>File: part details are stored in</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4)

Figure D.16: All file stores in the Data Flow method.

Figure D.13 shows all the data flows in the Data Flow method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Create New Part/Report Status/Clerk</strong></td>
<td>Create new part process writes details of new part number process scans parts file.</td>
</tr>
<tr>
<td><strong>Assign Part Number process scans parts file</strong></td>
<td>Assign Part Number process scans parts file.</td>
</tr>
<tr>
<td><strong>Validate Part/Assign Part</strong></td>
<td>Validate Part process scans parts file look for errors or status messages.</td>
</tr>
<tr>
<td><strong>Validate Part/Clerk</strong></td>
<td>Validate Part/Validate part details message to clerk if valid.</td>
</tr>
<tr>
<td><strong>Report Status/Clerk</strong></td>
<td>Report any error or status message from create new part process.</td>
</tr>
<tr>
<td><strong>Create New Part/Report Status/Clerk</strong></td>
<td>Create new part process writes details of new part number process scans parts file.</td>
</tr>
</tbody>
</table>

Create(1) Import(2) Edit(3) Delete(4) Find(5) Help(PF4)

Figure D.17: All data flows in the Data Flow method.