1

PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Reproduction in microform

Reproduction en microforme

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Canada

PegasusBase: A Run-Time Schema Extendible System

by

Bin Yuan

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Ontario, Canada, K1S 5B6
December 11, 1992

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Your file / Votre référence

Our file / Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84138-9

Canada

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

**PegasusBase:
A Run-Time Schema Extendible System**
submitted by Bin Yuan,
in partial fulfillment of the requirements for
the degree of Master of Engineering

Thesis Co. Supervisor

Thesis Co. Supervisor

Chair, Department of Systems and
Computer Engineering

Carleton University
January 5, 1993

Abstract

To meet the needs of run-time data structure extension in the Pegasus project, PegasusBase has been designed and implemented as a run-time extendible data structure system, a real time data manager and a communication tool for multiple processes. A special data model developed in PegasusBase to support data structure evolution is described in this thesis, and the algorithms based on the model are also presented and analyzed. The data model plays a key role in the system and can be readily extended into an object-oriented data model.

*This thesis is dedicated to my mother Menglan and my father Lifan who
have always encouraged my academic interests.*

Acknowledgments

I would first like to thank my advisors, Dr. B. Pagurek and N. Dawes, for their invaluable guidance and assistance, and for their confidence in me which kept me going. I would also like to thank Dr. A. R. Kaye who helped me get started in graduate studies and this project, and also Paul Kalab, Sami Iqneibi and all of the Pegasus group members for their suggestions and assistance in testing the computer programs and integrating the programs into the Pegasus diagnostic software.

List of Tables

Table 2.1	The definition of student.	13
Table 5.1	The comparison of different inter-process communication . .	33
Table 5.2	The class semaphores.	38
Table 5.3	The class ask_semph.	39
Table 5.4	The class members of PegasusBase.	44
Table 5.5	The structure definition of shm_inf.	50
Table 5.6	The definition of att_dfn.	53
Table 5.7	The structure of record_inf.	54
Table 5.8	The definition of structure r_att_node.	56
Table 5.9	The structure of record_att.	56
Table 5.10	Public interface.	60
Table 6.1	The definition of course.	75
Table 6.2	The structure of find_struct.	83
Table 6.3	A simple example in Pegasus.	85

List of Figures

Figure 2.1	Network management protocol architecture.	7
Figure 2.2	Blackboard architecture in Pegasus.	12
Figure 3.1	A class with four schema versions.	21
Figure 4.1	The general architecture of Pegasus.	23
Figure 5.1	The class PegasusBase.	29
Figure 5.2	A student database.	32
Figure 5.3	PegasusBase in Pegasus.	35
Figure 5.4	Current and history databases.	36
Figure 5.5	Multiple P operations.	41
Figure 5.6	The different views of PegasusBase.	42
Figure 5.7	The system models in the class PegasusBase.	46
Figure 5.8	Shared memory configuration.	47
Figure 5.9	PegasusBase in shared memory.	52
Figure 5.10	The definition of record R_5 in PegasusBase.	55
Figure 5.11	The organization of record attribute definition.	55
Figure 5.12	A record in PegasusBase.	58
Figure 5.13	Inserting a new attribute into the attribute library.	60
Figure 5.14	Allocate a new record.	61
Figure 5.15	An Example of Different Versions.	67
Figure 5.16	Overloaded constructors in PegasusBase.	69
Figure 6.1	The result after set().	78
Figure 6.2	Version relationships.	79
Figure 6.3	Versions of a student record.	80

Table of Contents

Abstract	ii
Acknowledgments	iv
List of Tables	v
List of Figures	vi
List of Abbreviations and Acronyms	x
Chapter 1 Introduction	1
1.1 Pegasus: a Multi-paradigm Diagnostic System	1
1.2 Special Pegasus Requirements	2
1.3 PegasusBase: A Run-Time Schema Extension System	3
1.4 Thesis Outline	5
Chapter 2 A Brief Review of Network Management	6
2.1 Network Management	6
2.1.1 Fault Management	8
2.1.2 Accounting Management	8
2.1.3 Security Management	8
2.1.4 Configuration Management	9
2.1.5 Performance Management	9
2.2 Fault Management and Related Work	9
2.3 The Pegasus Project	11
Chapter 3 State of the Art in Schema Extension	15
3.1 Conventional Databases	15
3.2 Next-Generation Databases	17
3.3 Object-Oriented Methods	18
3.3.1 Introduction to Object-Oriented Methods	18
3.3.2 Schema Evolution	19
Chapter 4 Database Requirements for the Pegasus Project	23
4.1 Schema Extension and Versioning	24
4.2 Fast Response Time	26
4.3 Sharing by Multiple Knowledge Sources at a Sun Workstation	27

Chapter 5 PegasusBase Design	28
5.1 Design Overview	28
5.2 Terminology	29
5.3 Shared Memory, the Communication Tool for Multiple Processes	31
5.4 Overview of the System	34
5.5 Semaphores	36
5.5.1 P and V operations	36
5.5.2 Semaphores in PegasusBase	37
5.6 The Design of PegasusBase	41
5.6.1 PegasusBase as a Class	43
5.6.2 The System Models	46
5.6.2.1 Shared Memory Information (ShmInf)	49
5.6.2.2 PegasusBase Attributes A	51
5.6.2.3 Record Information R	53
5.6.2.4 Record Attribute Definition $R_k.A$	54
5.6.2.5 Attribute Values ($\&R_k.A_i$)	57
5.6.3 PegasusBase Manager	57
5.6.3.1 Define a New Attribute	59
5.6.3.2 Allocate a New Record	61
5.6.3.3 Set a Value to a Record	61
5.6.4 Get an Attribute Value from a Record	65
5.6.5 Clear a Record	65
5.6.6 Create a New Version of an Existing Data Structure	66
5.6.7 Get the Parent Version	67
5.6.8 Overloaded Constructors	68
5.6.9 Destructor in PegasusBase	71
 Chapter 6 Data Structure Extension and Versioning in PegasusBase	 73
6.1 Build the Sample Database	73
6.2 Extend the Data Structure	74
6.2.1 Define New Attributes in A	74
6.2.2 Set Values to Records	76
6.3 Versioning in PegasusBase	78
6.4 Retrieving Information from PegasusBase	82
6.5 A Simple Example in Pegasus	84
6.5.1 Find out What Is Broken	86
6.5.2 Add a New Node Type	87
6.5.3 Create a New Version	88

Chapter 7 Conclusions	89
7.1 Contributions	89
7.2 The Use of C++	90
7.3 Future Work	91
References	93

List of Abbreviations and Acronyms

CODASYL — the Conference on Data Systems Languages

DBMS — DataBase Management System

DBTG — the Database Task Group

DCL — Data Control Language

DDL — Data Definition Language

DML — Data Manipulation Language

KS — Knowledge Source

LAN — Local Area Network

OODB — Object-Oriented DataBase

OODBMS — Object-Oriented DataBase Management System

OSI — Open System Interconnection

TRIO — Telecommunications Research Institute of Ontario

WAN — Wide Area Network

Chapter 1

Introduction

1.1 Pegasus: a Multi-paradigm Diagnostic System

Large communication networks are becoming more complex and crucial to business, enterprise and society. As the networks grow in size and intricacy, effective management which ensures timely and reliable delivery of information through them becomes increasingly difficult. Network fault management which diagnoses, locates and predicts faults in networks is therefore also getting more complex and difficult. In the communication industry, there is great interest in and considerable pressure for the development of common techniques and strategies to effectively manage such complex networks. Pegasus, a project sponsored by the Telecommunications Research Institute of Ontario (TRIO), is a multiple paradigm approach which detects many faults before they become serious and finds the appropriate changes necessary to correct the failure [1].

To reach the goal of automating the process of diagnosis and improving both the operational performance of existing networks and the potential performance of future networks, Pegasus employs several cooperating intelligent diagnostic agents to detect and locate faults for the integrated communication networks [2]. The cooperating agents are organized through a blackboard architecture in a UNIX environment and currently

include:

- **BB**, a blackboard system which serves as a global database and mailbox, and manages the communication and access control among the agents [3];
- **Dragon**, a network simulator which simulates wide area networks (WANs) [4];
- **BANES**, a probabilistic reasoner which collects evidence of network faults from the status messages generated by data collectors [5];
- **END**, a rule-based expert network diagnostician which uses rules acquired from human expert network operators to diagnose network faults [6];
- **LEAPR**, a neural network analyzer which recognizes certain types of problems during their early stages and reacts before they become critical [7].

1.2 Special Pegasus Requirements

In Pegasus, multiple knowledge sources are organized through a blackboard architecture in a UNIX environment. The blackboard system serves as a global database, controls communication among knowledge sources, and manages status messages generated by data collectors and hypotheses proposed by knowledge sources. The data managed by the blackboard can only be those predefined by the system, and any changes to the data structure may lead to the need to rewrite the source code and temporarily shut down the system. However, Pegasus must deal with networks which change dynamically with

time, for example, as new devices are added, old nodes are removed and new LANs are incorporated into the network. This means that the data in Pegasus may need to be viewed differently at different times by different knowledge sources, and the attributes of the data may change as new devices are incorporated into the network. The database used by Pegasus should therefore have the ability to adapt to changes in the WAN and provide all of the services required by Pegasus. However, a blackboard system cannot cope with networks that change dynamically and still provide the required services. A special data manager or database system that has the ability to deal with the changes of a WAN, provides run-time schema or data structure extension, real time response and supports multiple simultaneous users is therefore required by Pegasus.

1.3 PegasusBase: A Run-Time Schema Extension System

PegasusBase has been designed, tested and implemented to satisfy the special requirements for a database system in Pegasus.

PegasusBase is a run-time schema extension system with real time response, and it can be shared by several knowledge sources running on a single workstation in the UNIX environment. The entire system is designed with object-oriented methods and implemented in C++. The main contribution of this work is the design and testing of the efficient run-time extendible schema method.

The data model used in PegasusBase is specially designed to satisfy the requirements of the Pegasus project. It employs five blocks: shared memory information, PegasusBase attributes library, record information, record attribute definition and attribute values. The shared memory information block stores the memory management information, and the other four blocks act as the schema manager. The data structure of a record extends as new attributes are added and as the values of the attributes are set. The data model itself was written using an object-oriented approach and can be extended into an object-oriented database model by adding two more levels which manage methods in a class and inheritance of classes.

In addition to the need for run-time data structure extension, there is another requirement: the ability to access the history data of a network object by some knowledge sources, such as those using case-base reasoning. Therefore, two parts of PegasusBase are used: one is the current database which holds the most recent messages and is shared by the agents accessing it, and the other is a history database which loads the required information from disk storage to memory as required and is accessed only by its owner. The current database provides run-time data structure extension, real time response and a database which may be shared. The history database provides the versions of a record and maintains the history

of the different states of a specific network object; such a history can be used by knowledge sources which require it in order to formulate and test a hypothesis.

1.4 Thesis Outline

The thesis is organized as follows. Chapter 2 has a brief review of network management. In that chapter, network management is summarized in general terms, the relevant aspects of fault management are discussed and the blackboard of the Pegasus project is introduced. Chapter 3 describes the state of the art in schema extension. This chapter presents the schema extension in both conventional and next-generation database management systems and describes object-oriented methods in their field. Chapter 4 defines the problems which this thesis addresses. The three problems discussed in this chapter are schema extension and versioning, the response time of the database and its shareability with multiple knowledge sources at one workstation. Chapter 5 describes the design of PegasusBase in detail and Chapter 6 describes how PegasusBase extends the data structures. Chapter 7 discusses the contribution of this thesis to the Pegasus project and the work which may be undertaken in the future.

Chapter 2

A Brief Review of Network Management

2.1 Network Management

The rapid development of communication technologies permits information to be transmitted and received at high speed over great distance. Voice-mail, electronic mail and the FAX increase both cooperative interpersonal activities and the requirement for advances in communications. Today, the range of networks includes not only computing centered around mainframes but also local area networks (LANs), wide area networks (WANs), international networks, voice (public and private) networks and packet data networks [8]. Computer networks become larger, often involving thousands of elements, more heterogeneous, supporting a variety of devices and protocols, and more complex, involving subtle interactions and relationships among components. Networks have been the information highway in business, enterprise, universities and almost every aspect of our society. To ensure smooth efficient and reliable operations of networks, there must exist effective management tools. Thus, integrated network management is becoming a key issue in the communication industry [9][10].

An integrated network management system is a complex application system that lets people integrate, control and manage heterogeneous networks involving a multiplicity

Figure 2.1 Network management protocol architecture.

of vendor process and communication products [11]. It extends the well-known Open System Interconnection (OSI) seven-layer model in networks [12]. Figure 2.1 depicts network management as a specific seven-layer model.

A model for OSI management is described in the Management Framework [13] and in the System Management Overview [14]. The authors describe a basic architectural model for OSI management and define management requirements by identifying the following five specific management function areas [15]:

- Fault Management;
- Accounting Management;
- Security Management;

- Configuration Management;
- Performance Management.

2.1.1 Fault Management

Fault management detects, isolates and corrects any abnormal operation of the OSI environment. Its functions are to maintain and examine error logs, accept notifications that errors have been detected and respond to these notifications, trace and identify faults, carry out diagnostic tests and correct faults.

2.1.2 Accounting Management

Accounting management supports the cost allocation algorithms developed by other groups. Development of the proper interfaces for flow of cost information into and out of the finance department is essential to corporate success [16].

2.1.3 Security Management

Security management ensures that corporate information which may be accessed via the communications domain remains secure. A close working relationship with the security department is essential in establishing the criteria and cost for the protection of information. The encryption password access and audit procedures are the responsibility of the security department.

2.1.4 Configuration Management

Configuration management dynamically tracks the physical configuration of the equipment, both hardware and software, and changes in the network topology. When resources or other attributes are added, they are collected by the nearest server or directory service. The task of keeping the network information database up to date with changes requires an automated approach. Since the database feeds several network functions and is the recipient of other data items, it must be a very robust and reliable system.

2.1.5 Performance Management

Performance management uses inputs (e.g. link error rate, service priority, etc.) from the configuration files and applies management criteria in determining the degree of the severity of a fault, and the priority which should be assigned to restoration procedures; these may change over time.

2.2 Fault Management and Related Work

Reliability has always been one of the fundamental issues in computer networks. As mentioned above, fault management is a set of activities which enable the detection, isolation and correction of network faults [17] [18]. However, as the size and complexity

of networks grows and more sophisticated hardware and software is used, network fault isolation and diagnosis becomes far more complex.

Many vendors have developed some form of fault management for their network management products [19]. For example, IBM's NetView has monitoring, alert reporting, problem determination and isolation, and tracking functions [20], DEC's phase V network management, and AT&T's Unified Network Management Architecture (UNMA) closely follows the ISO network management standards, including fault management [21].

Numerous research papers have dealt with the problems of detection, isolation, diagnosis and recovery of network faults. Among them, artificial intelligence techniques play a very important role. The terminologies, such as expert system [22][23], knowledge-based system [24], Neural network [25], machine learning [26] and blackboard system, are used so frequently that we can find references to them in many papers. Paul Hong and Prodip Sen proposed a reasoning technique which uses uncertainty to detect heterogeneous network faults [27]. In this approach, a network proposition is evaluated with a belief metric, which uses a pair of support measures or votes, such that potential "*don't know*" and "*contradicting*" conclusions can be expressed and derived. Frontini et al. described a knowledge-based system which was developed for monitoring and locating fault in a wide area network [28]. This system "*incorporated some of the network*

operators' and manager's knowledge on localizing faults " by using the structure of the network and correlating events. It was implemented by using a blackboard architecture.

All of the authors referenced above showed that some artificial intelligence techniques have moved out of the laboratory. Along with them, a whole new style of automation system design and programming is being introduced into computer network management. Pegasus [29], a new fault diagnostic system with several cooperating intelligent agents, is a successful example of this.

2.3 The Pegasus Project

The Pegasus system is designed to diagnose faults in communication networks. It has a variety of reasoning methods available and is based on the blackboard architecture (see Figure 2.2).

A blackboard is a shared data structure used for communications by different knowledge sources. A common pool of information stored in a blackboard can be used and modified by the knowledge sources. In this way, several reasoning agents can cooperate with each other in order to solve network management problems.

The most efficient method for implementing the blackboard in a UNIX environment is to use shared memory applications which rely on virtual memory management [3].

Figure 2.2 Blackboard architecture in Pegasus.

Shared memory allows more than one process at a time to attach a segment of physical memory to the virtual address space of the process. A blackboard can be viewed as such a segment of memory which is attached to more than one process.

Shared memory is a means by which the blackboard can be implemented. A region

of shared memory can be viewed as a blackboard, and the contents (data or records) on the blackboard are the shared knowledge. How the data is organized on the blackboard is very important to the knowledge sources using the shared memory, because the knowledge sources must know where to find the data on the blackboard and how to access it. Data with predefined structures can be easily fitted onto the blackboard or the region of shared memory by assigning a predetermined position to each of the data items. After the data is written to the blackboard, the knowledge sources can read or rewrite the data by accessing the same position.

For example, we have a structure named *student* defined in Table 2.1

Table 2.1 The definition of *student*.

FIELD NAME	FIELD TYPE	DESCRIPTION
student-name	string	The field for student name
age	integer	The age of the student
sex	character	'M' for male and 'F' for female

Assume that the shared memory is attached to the virtual memory space of the process at address 1000 and is large enough to hold 100 *student* data items. Because the size of the structure is predefined, it is easy to give each of the data a fixed space on the blackboard. If the first data item of *student* begins at address 1000 of the virtual memory, then the second one can be at $1000 + \text{sizeof}(\text{student})$, the *i*th one at

$1000 + (i - 1) * \text{sizeof}(\text{student})$. This is called determined organization with which every process can access the data and its fields in the shared memory easily. However, those records which must be defined or whose structure must be modified at run-time cannot use the blackboard system described above.

Chapter 3

State of the Art in Schema Extension

3.1 Conventional Databases

Databases are large collections of structured data stored in one or several computers in order to serve several data processing applications. A database management system or DBMS provides an environment, both convenient and efficient, for use in retrieving information from and storing information in the database.

The database technology for information systems has undergone four generations of evolution; the fifth generation database technology is currently under development [30]. The four generations are file systems, hierarchical database systems, CODASYL databases [31] and relational databases. The second and third generation systems realized the sharing of an integrated database among many users within an application environment. The lack of data independence gave rise to the fourth-generation database technology, relational databases. The object-oriented database is one of the best known "next-generation" database systems. It is built on object-oriented concepts, and directly supports the needs of the applications that create and manage objects that have the object-oriented semantics, namely, programming languages or applications designed in an object-oriented style.

Conventional database systems, relational and past-generation database systems, provide database languages to allow application programmers or end-users to define and manipulate the database. A conventional database consists of two main components: a data definition language (DDL) and a data manipulation language (DML) [32].

The DDL is used to define the database schema and also to give some details about how to implement this schema in the physical devices used to store the data [33]. In relational databases, the schema of a database is the specification of a set of relations: the name of each relation, the name of each column of a relation, the domain (data type) of each column, and integrity constraints on the domain. The definitions, which can be described as metadata about the data in the database, are expressed in the DDL of the DBMS and are maintained in a compiled form, known as a data dictionary, directory or system catalog. The data dictionary contains information about the data stored in the database and is consulted by the DBMS before any data manipulation.

The DML is the language used to manipulate data in database after the schema is defined.

In relational databases, a relation is an abstraction of some portion of the real world that is being modeled in the database, and since the real world changes with time, the data items of a relation also vary over time. Thus, data items may be added, deleted, or updated. However, the relation schema itself does not change, at least until the database

is reorganized. Even though a number of approaches have been proposed to deal with the schema update [34] [35] [36], there appears to be no consensus forthcoming.

In hierarchical and network databases, the types and the relationships between record types must be predefined, and these predefined structures and relationships are usually implemented at the physical level with the use of linked structures. The application programs depend heavily on these physical structure. This results in faster access to related records than is possible in the relational case, but also results in a lack of data independence that makes it more difficult to update the schemes in hierarchical and network databases.

3.2 Next-Generation Databases

Currently, there are at least two proposed approaches to a transition from fourth-generation database technology to the fifth-generation technology: extended relational databases and object-oriented databases. The former is trying to provide better support for complex objects, such as CAD/CAM, geographic information systems and software engineering, than what is offered by the relational model and still retain such important features as a powerful, user-friendly, data manipulation language [37]. For example, POSTGRES [38], a well-known next-generation database system, is based on the extended relational approach. The second approach adopted in the Orion system [39]

and a number of other systems, such as *O₂* [40] and GemStone [41], starts with an object-oriented data model and a database language that captures the model, and can extend them in various ways [42].

One of the major advantages of the object-oriented database is that its data model can represent much more of the semantics of data directly as a form of a database schema than the relational model can [43].

3.3 Object-Oriented Methods

3.3.1 Introduction to Object-Oriented Methods

Object-oriented techniques have drawn considerable interest in the last few years. The term “object-oriented” has been used to describe almost all of the aspects of computing, including object-oriented languages, object-oriented design and development, object-oriented programming, object-oriented databases, object-oriented computer architectures, etc. [44].

An object-oriented approach to programming is based on the concepts of encapsulation and extensibility. The traditional view of software systems is that they are composed of a collection of *data* that represents some information and a set of *procedures* that manipulates the data. However an object-oriented system has only one single

type of entity, the *object*, instead of the two types which represent information and its manipulation independently. Object-oriented programming encapsulates some data and procedures/methods in an object. The data is the state of the object, and the set of procedures determines the behavior of the object.

An object-oriented approach to programming offers extensibility in two ways: behavioral extension and inheritance. The behavior of an object may be extended by simply including additional programs. An object-oriented approach further promotes extensibility through reuse or inheritance. The behavior, and even the attributes, defined for an object may be reused in the definition of more specialized objects, that is, they are inherited by the new objects. Furthermore, one may define additional behavior and attributes for an object, and even redefine some of the inherited behavior and attributes.

Some advanced systems, such as Orion [39], O_2 [40], GemStone [41], Avance [45] and Iris [46], have taken the approach of designing an architecture to directly support the object-oriented paradigm. However, none of them can offer schema versioning and real time services as discussed below.

3.3.2 Schema Evolution

The extraction of data object type description from application program into a single repository is called a schema or a database schema. Updating the schema is

an important facility for object-oriented databases. However, updates must not result in inconsistencies either in the schema or in the database.

Won Kim has classified the dynamic schema modification in an OODB into two dimensions [47]. One is called single schema modification and the other is called schema versioning. Single schema modification is the direct modification of a single logical schema and schema versioning is the versioning of a single logical schema. Schema versioning makes different views of the database visible to the users using different versions of the schema.

Schema modification may generate two problems: how can the history of the objects be preserved as they are modified and how can changes to the schema by one user be prevented from affecting all other users' views of the database. Schema versioning was proposed to solve these problems [48]. The system will manage more than one logical schema for one common database, and present different views of the database through different versions of the schema. For example, if a user wishes to add or drop an attribute to or from a class in one version of the schema he may create a new version of the schema that will reflect the schema change. If the user chooses to access the class through the previous version of the schema he will be able to see the instances of the class that existed before the new version of the schema.

In Figure 3.1, the schema of class *student* has four versions and each version contains

Figure 3.1 A class with four schema versions.

several instances of that version. When we choose to access version V2, we can see that this class has three types (*name*, *address*, *phone*), and one operation (*method1*). Several instances of this version exist. If we go back to version V1, there are only two types (*name*, *address*) and one operation (*method1*) in this version. *Phone* does not occur in version V1, therefore, we cannot access *phone* unless we access the version which contains this attribute (i.e.: version V2 or V3).

The model of single-schema modification for an object-oriented database was established in the systems Orion, Avance, and GemStone. However, no database system,

object-oriented or otherwise, supports schema versioning so far [30]. Schema versioning will be a new and advanced feature for a database system. PegasusBase provides this capability.

Chapter 4

Database Requirements for the Pegasus Project

Pegasus is a multi-paradigm diagnostic system for wide area communication networks. It employs several cooperating intelligent diagnostic agents to detect and locate faults for wide area networks. Figure 4.1 represents the general architecture of Pegasus.

There are some special requirements for the database used in Pegasus, which are:

- Schema extension and versioning
- Fast response time
- Simultaneous access by multiple knowledge sources (processes) at a Sun workstation

Figure 4.1 The general architecture of Pegasus.

4.1 Schema Extension and Versioning

The overall design of the database is called the database schema. A database schema is specified by a set of definitions and results in a special file called the data dictionary. A data dictionary, also called meta-data, contains the information about the structure of the database, and is consulted before actual data is read, added, modified or deleted in the database system. It is heavily used and plays a very important role in the database system. All of the application programs are dependent upon it.

In the Pegasus environment, many agents share a single database and cooperate to diagnose the faults in an integrated network. An integrated network is a network often involving thousands of elements, supporting a variety of devices and protocols, combining hardware and software produced by a large number of vendors and, most importantly, having a dynamically changing topology. In this type of network, some unexpected devices or newly developed communication equipment may be added and routers, LANs, bridges, workstations or other nodes may be removed or added. In order to deal with this ever changing network topology, the Pegasus system will have intelligent agents which can cope with the changing network environment. This means that the knowledge agents employed by the Pegasus system might view the data in the database differently at different times. The schema needs to be changed to reflect the changing topology. It is highly desirable for the data-

base system to be able to dynamically extend the schema or data structure so that the system can deal with the changes of the network without forcing Pegasus either to shut down or to incur a significant performance penalty. This defines the requirement for schema extension and versioning, particularly in the data structure portion.

For example, a node in a network called bCARc1 connects to nodes bMTLc2 and nTORc1. bCARc1 is a router located in Ottawa. Now bCARc1 changes its type from router to Brouter, a new type with more features than a router. To describe a Brouter, a new complex attribute is created. Since bCARc1 is a type of Brouter now, its record needs the new attribute of Brouter. To diagnose the node bCARc1, an intelligent agent needs to know how bCARc1 is configured and how it behaved before and after the change. In this example, we must keep the different versions of the record in addition to extending the record structure properly.

In traditional DBMSs, the schema cannot be extended unless the application system is shut down or the whole database is reorganized. Furthermore, once transaction updates have been committed and permanently stored, the previous values of data are usually discarded. No versioning of the data is possible.

Object-oriented databases, like Orion, *O₂*, Avance and Iris, can provide versions of objects (different values but the same structure) and schema extension (without keeping

the previous schema), but none of them can offer schema versioning (keeping the previous schema in different version).

4.2 Fast Response Time

An integrated network includes bridges, LANs, routers, and thousands of workstations and personal computers. In order to diagnose problems in this type of network, the information about the network must be collected and processed quickly. In the BNR/NT network, on which the Pegasus system is based, several computers, called data collectors, are designed to monitor the important parts of the network. If any device monitored by a data collector malfunctions, an appropriate status message is sent to the network operation centre. This message is then passed to a database through the blackboard system and is used by the intelligent agents to carry out the diagnosis.

To develop, test and analyze the Pegasus system, a large-scale and high-speed network simulator, Dragon [4], is used. Dragon can simulate a very large and extremely busy WAN with line speeds ranging from 56 Kb/s to 100Mb/s, and supporting at least 100 routers, 255 LANs, 500 bridges and several thousand workstations faster than real time. To deal with the data entering the system at such speeds, the database must have at least the capacity to deal the information sent by Dragon. The response time of a database used in Pegasus becomes very critical. The correctness of computation or

processing in Pegasus depends not only on the logical results, but also on its timeliness. From this point of view, we can say that a real-time database is required in the Pegasus project.

An object-oriented database can support the complex requirements of encapsulation, inheritance and polymorphism, but so far none can offer a real-time service.

4.3 Sharing by Multiple Knowledge Sources at a Sun Workstation

Pegasus employs several cooperating diagnostic agents and runs on a workstation under a UNIX environment. To support the multiple agents or knowledge sources, a database must have the capability of being shared by multiple processes or knowledge sources at the same workstation.

The current OODBMSs cannot be shared by multiple processes at the same workstation. Orion has three systems, Orion-1, Orion-1sx and Orion-2 [30]. Orion-1 is a system designed for a single processor. Orion-1sx is an object server that manages a persistent database on behalf of a number of client workstations which do not have their own persistent database. Orion-2 is a fully distributed database system. O_2 is distributed as a server/workstation configuration. The workstation component is memory-based single user, while the server version is a disk-based multi-user system [40].

Chapter 5

PegasusBase Design

5.1 Design Overview

PegasusBase is a database manager designed for Pegasus. It uses object-oriented methods, communicates through shared memory and controls access to the shared memory by semaphores. PegasusBase allows run-time data structure extension, has real-time response and is integrated with various intelligent agents running on a single workstation.

PegasusBase is implemented by object-oriented methods. However, it is not an object-oriented database yet. The design abstracts the system data and encapsulates them and system operations as a class. The class includes some objects of other classes, class *semaphore* and class *ask-semph*. Polymorphism is also used in the design. Inheritance, however, is not used, because the data model is stored in shared memory. Figure 5.1 shows the class PegasusBase.

In PegasusBase, every record has a record header which points to the record definition. Record definitions are lists of field descriptors. Every field descriptor points to an attribute definition and to the attribute value. The system design will be described in detail in the following sections of this chapter.

Figure 5.1 The class PegasusBase.

5.2 Terminology

Some terminology used in the design can be defined as follows:

- **Attribute set (A)** — All of the attributes used in PegasusBase form a set called attribute set A . $A = \{A_1, A_2, \dots, A_i, \dots, A_m\}$, owned by the system.
- **Record set (R)** — All of the records in the system form a record set R , $R = \{R_1, R_2, \dots, R_k, \dots, R_n\}$.
- **Record element (R_k)** — A record that belongs to the record set, that is, $R_k \in R$.
- **Record attribute set ($R_k A$)** — A subset of A , $R_k A \subseteq A$, which defines the record structure of record R_k .
- **A record attribute ($R_k A_i$)** — A record attribute which is an element of A . $A_i \in A$.

and is also an element of $R_k A$, that is, $A_i \in R_k A$, and also called $R_k A_i$. The index i here does not refer to the order in the list $R_k A$, it is the attribute index or name from set A .

- An attribute value ($\&R_k A_i$) — The value of $R_k A_i$.
- The number of elements in a set ($|S|$) — S is any set of attributes and $|S|$ is the number of elements in S .

For example, a student database holds ten attributes: student name, student number, telephone number, mailing address, start time, course1, course2, course3, course4 and course5. The attribute set is $A = \{\text{student name, student number, telephone number, mailing address, start time, course1, course2, course3, course4, course5}\}$, where

$A_1 = \text{student name}$

$A_2 = \text{student number}$

$A_3 = \text{telephone number}$

$A_4 = \text{mailing address}$

$A_5 = \text{start time}$

$A_6 = \text{course1}$

$A_7 = \text{course2}$

$A_8 = \text{course3}$

$A_9 = \text{course4}$

$$A_{10} = \text{course5}.$$

The students in the database form a record set R . If there are 1,000 students in the set, then

$$R = (R_0, R_1, \dots, R_k, \dots, R_{999}).$$

Let student k called *John Smith* be a record element R_k having attributes: student name, student number, telephone number, mailing address, start time, course1 and course4. Then we have:

$$R_k.A = \{\text{student name, student number, telephone number, mailing address, start time, course1, course4}\}.$$

Also we have $R_k.A_9 = \{\text{course4}\}$, an element of $R_k.A$; the value of *student name* of R_k is $\&R_k.A_1 = \text{John Smith}$, and the number of elements in $R_k.A$ is 7, that is, $|R_k.A| = 7$. This example is shown in Figure 5.2.

5.3 Shared Memory, the Communication Tool for Multiple Processes

In the UNIX System V, processes can communicate directly with each other by sharing parts of their virtual address space called shared memory. Because shared memory is a segment of physical memory which is attached to more than one process's virtual address, communication among many processes can be easier and faster by using

Figure 5.2 A student database.

shared memory than if other facilities such as 'pipe' and 'message' are used (see Table 5.1). To prevent inconsistencies and collisions, an outside protocol or mechanism, such as a semaphore, can be used when write access is allowed for more than one process.

Using System V shared memory, several system calls are offered to create, attach

Table 5.1 The comparison of different inter-process communication

Inter-process Communication Methods	Description	Time Efficiency
Shared Memory	allow processes to share parts of their virtual address space	high
Messages	allow processes to send formatted data streams to arbitrary processes	fair
Pipes	communicate by writing to and reading information from files	low
Messages	use file I/O model and are preferred when a service has both local and network clients	low

and detach the memory. The system call *shmget* creates a new region of shared memory or returns an existing one, the system call *shmat* logically attaches a region to the virtual address space of a process, *shmdt* detaches a region from the virtual address space of a process, and *shmctl* manipulates various parameters associated with the shared memory. After shared memory is attached to a process, it becomes part of the virtual address space of the process and is accessible in the same way as other virtual addresses; no system calls are needed to access data in shared memory.

In PegasusBase, the current database is stored in shared memory so that any changes in it can be directly accessed by other knowledge sources.

5.4 Overview of the System

PegasusBase is a database system designed to fulfill the requirements of the Pegasus system. With object-oriented methods, PegasusBase was written in C++ under the UNIX environment and can be easily integrated with any other agents that are written in different programming languages. In Pegasus, the agents share the messages stored in PegasusBase through the blackboard system (see Figure 5.3).

PegasusBase offers two types of information bases. One is called the current database, where the most up-to-date information is stored, and the other is called the history database, where the information stored on a disk file is loaded only when needed. Both of them share the same technique.

The current database is used to collect current data from the network which is then used by the knowledge sources. It is organized in the shared memory so that the information in the database can be accessed by every agent in Pegasus.

The history database is used by some knowledge sources when historic information is required. In the current implementation, if the current database is completely filled,

Figure 5.3 PegasusBase in Pegasus.

contents of the shared memory are stored in a disk file to form an old page of PegasusBase. When a page is selected, it is loaded into the history database and used by the knowledge source requiring that page. Every knowledge source can have a history database, but each history database is local to one knowledge source only. Figure 5.4 shows the two databases in Pegasus.

Because both the current and history databases share the same technique except that history database does not need any mechanism to prevent collisions and inconsistencies, only the current database is described in this thesis. The models used in the current

Figure 5.4 Current and history databases.

database are also used in the history database. When the term PegasusBase is used, it refers to the current database, unless stated otherwise.

5.5 Semaphores

5.5.1 *P* and *V* operations

Semaphores are integer-valued objects which have two atomic operations defined for them: *P* and *V*. The *P* operation decreases the value of a semaphore if its value

is greater than 0, and the *V* operation increases its value. Because the operations are atomic, at most one *P* or *V* operation can succeed on a semaphore at any time. The semaphore system calls in the UNIX System V are the implementations of *P* and *V* operations. In these calls, several operations can be done simultaneously by the kernel; no other processes can adjust the semaphore values until all the operations are done. If the kernel cannot do all the operations, it does not do any; the process sleeps until it can do all the operations.

5.5.2 Semaphores in PegasusBase

In PegasusBase, multiple processes can read at the same time from the shared memory, but only one process can write to the shared memory. Semaphores are used to synchronize executions and maintain data consistency in shared memory. The initial semaphore value is set to 10. There are two *PV* operations defined in PegasusBase, **READ-ONLY** and **WRITE**. **READ-ONLY** operation has value 1. Up to ten such operations are allowed in the shared memory at the same time. **WRITE** operation decreases a semaphore value by 10, which means that there is only one such operation that can be done at any one time and any other operations are forbidden.

For example, two processes *P1* and *P2* are reading from shared memory; the semaphore value is 8 because each **READ-ONLY** operation decreases the semaphore

value by 1 before the READ can actually be executed, and increments the semaphore value by 1 after the READ is done. If another process, P3, wants to write to the shared memory while the first two processes are still reading, then P3 sleeps because the kernel cannot complete the *P* operation. P3 will wake up after P1 and P2 finish reading the shared memory.

In addition to the READ-ONLY and WRITE semaphore operations, there is an *intl_lock* operation. When a process obtains the authority to create and initialize the current database (shared memory), an *intl_lock* operation is issued to lock the shared memory so that no other process can access it. After the initialization, the process sets the semaphore value to the initial value 10.

Table 5.2 The class *semaphores*.

PRIVATE	NAME	TYPE	DESCRIPTION
	semid	integer	Semaphore ID
PUBLIC	semaphores	constructor	Initialize the class
	intl_lock	operation	Lock shared memory while it is initialized
	set_intl_semph	operation	Set the initial value of a semaphore
	get_semid	operation	Return the semaphore ID

Two cooperating classes are enclosed in semaphore operations: *semaphores* and

ask_semph. Class *semaphores* is used to get information from the semaphore, such as semaphore ID, and to set the initial value of the semaphore. Table 5.2 illustrates the definition of class *semaphores*. Class *ask_semph*, shown in Table 5.3, is used to perform the *PV* operations and is a friend class of *semaphores*. This means that *ask_semph* can access a member of *semaphores* directly. The definition of *ask_semph*, shown below, has two overloaded constructors, one destructor and an operation called *sem_value()* which returns the current semaphore value.

Table 5.3 The class *ask_semph*.

PRIVATE	NAME	TYPE	DESCRIPTION
	semval	integer	Semaphore value
	nsem	static integer	The number of semaphore operation request
	sem_op	operation	Semaphore operation
PUBLIC	ask_semph	constructor	Overloaded constructor initializing nsem
	ask_semph(int)	constructor	Overloaded constructor asking for a semaphore
	~ask_semph	destructor	Clear the semaphore when the operation is finished
	sem_value	operation	Get the value of the semaphore

In PegasusBase, public operations can be grouped into two types: READ-ONLY and WRITE. Any operation accessing the shared memory without changing the information is grouped in READ-ONLY and the other operations belong to the WRITE group. Every

public operation must include a *P* operation to ask for the right to access the shared memory. However the operations may call each other, for example, in public operation M1, another public operation M2 is called, as a result of this, more than one *P* operation is involved. To solve this problem, a variable *nsem* is used in class *ask_semaphore()*.

Normally, each object of a class has its own copy of the data variables of the class. However, it is possible to define members which are shared among all objects of the class and once the values are changed by one of the objects, other objects can get the results. This type of member is called a *static* data member. In *ask_semaphore*, *nsem* is a static data member which is shared by all objects of the class.

Nsem is set to zero by the in-line constructor *ask_semaphore()* which is used only once when PegasusBase is initialized. In each entry of the public operation of PegasusBase, there is an object of *ask_semaphore* declared by using the constructor of *ask_semaphore(int op)*. The constructor increments the data *nsem* by 1. If and only if *nsem* is 1, the *P* operation will be issued, therefore there is no duplex *P* operation in the system.

When an object of *ask_semaphore* is no longer needed, the destructor of *ask_semaphore* will decrease *nsem* by 1, and a *V* operation is issued if the access to the shared memory is finished.

For example, there are three public methods/operations defined as M1, M2 and M3 in a system using shared memory, shown in Figure 5.5. If M1 is executed, the three

objects, O1, O2, and O3, of *ask_semaphore* are all created, because M1 calls M2, and M2 calls M3. But only the first object O1 issues the *P* operation, because only O1 has *nsem* = 1. When M3, M2 and M1 are finished, the last object remaining in the system does the *V* operation. Only one pair of *PV* operations is done when any method accesses the shared memory, no matter how many methods are nested.

Figure 5.5 Multiple *P* operations.

5.6 The Design of PegasusBase

Virtually every program has two main ingredients: data and the operations used

to process the data. We can specify how data is to be represented by the use of data types and how data is to be processed by the use of functions. This is the conventional programming style called the function-oriented method. In PegasusBase, we use an object-oriented method to design the system, to merge data and functions together into a single framework.

PegasusBase is defined as a class rather than as a conventional database system. Even though using PegasusBase is as easy as the built-in types, PegasusBase has some of the basic functions of a DBMS (DataBase Management System).

Figure 5.6 The different views of PegasusBase.

Class PegasusBase has two ingredients: system data and related operations as

shown in Figure 5.6. The system data corresponds to the five models which determine the database organization, the storage distribution, the access strategies and schema management. The related operations, also called the PegasusBase Manager, offer the basic DBMS functions based on the system data.

5.6.1 PegasusBase as a Class

A data model is a logical organization of the real-world objects (entities), constraints imposed on them, and relationships among objects. The data that captures object-oriented concepts is an object-oriented data model. An object-oriented database is a collection of objects whose behavior, state and relationships are defined in accordance with an object-oriented data model. An object-oriented database system is a database system which allows the definition and manipulation of an object-oriented database. PegasusBase itself is an object-oriented data model, and also an investigation of object-oriented database architectures for network management and diagnosis.

PegasusBase is designed as a class which can manage the current database in shared memory and history database in local memory. It includes a semaphore control protocol of the shared memory to support data consistency, a data schema manager to allow run-time schema extension and a file manager to save memory data to disk to form the history database file and load the file to a local history database for a knowledge source.

The elements in PegasusBase are listed in Table 5.4.

Table 5.4 The class members of PegasusBase.

NAME	TYPE	DESCRIPTION
shm_state	state	This is the shared memory state. Only one shared memory is allowed.
shm_id	integer	This is an ID of the shared memory.
local_state	state	Local_state is the state of the history database.
semph	object of class semaphore	This is an object of class semaphores. It create and initializes the semaphore for the shared memory.
sem_id	integer	Semid is the ID of the semaphore.
*RecordInf	pointer	It is a pointer which points to the address of the record information model R .
*RecordAtt	pointer	It points to the address of the record attribute definition model $R_k A$.
*DataStart	pointer	It points to the attribute value area $\&R_k A_t$.
*Attributes	pointer	It points to the address of PegasusBase attribute model A .
*ShmInf	pointer	It points to the shared memory address which is also the address of shared memory information model $ShmInf$.
type	char	This is the type of the database. If type = 'I', then the database is a history database else type='s', the database is a current database.

Table 5.4 (Continued) The class members of PegasusBase.

PegasusBase()	constructor of current base	This is the constructor which creates and attaches the shared memory to the process and initialize the current base in shared memory.
PegasusBase(old_page)	constructor of history base	This is the overloaded constructor which initializes the history database for the process.
~PegasusBase	destructor	When an object of PegasusBase is out of scope, the destructor clears everything created by the object.
management functions		See 5.5.3 PegasusBase Manager.

In Table 5.4, there are five pointers pointing to PegasusBase system models which are the key parts of the system, and all of the methods in the system are based on them. It can be seen from Figure 5.7 that the messages or records in PegasusBase . organized and stored in the five system models. The class PegasusBase is an interface that manages the records in the models for users. How the system models are organized and operated is discussed next.

Figure 5.7 The system models in the class PegasusBase.

5.6.2 The System Models

The system models in PegasusBase are used or operated by the PegasusBase manager to provide central control of records and processes (knowledge sources) accessing that record. The system data stored in the shared memory can be divided into five models shown in detail in Figure 5.8.

The five models are:

1. Shared memory information area (ShmInf);

Figure 5.8 Shared memory configuration.

2. Record information (R);
 3. Record attribute definition ($R_k A$);
 4. Attribute value ($\&R_k A_i$);
 5. PegasusBase attribute definition (A).
1. **Shared memory information model (ShmInf)** is the shared memory information centre and resides in the lowest address of the shared memory. The first step for

PegasusBase in retrieving, updating or adding a record is to go through this model to get the associated information and perform the operations. Any changes in the shared memory are recorded in this model. *ShmInf* keeps the shared memory information and makes the organization of the database consistent. When the shared memory is created, the information about the memory is stored in *ShmInf*. Any process or knowledge source can get the information through the PegasusBase Manager and access the records. Any changes in the other four models will be written into *ShmInf* in order to preserve the consistency of PegasusBase and to make the data available to multiple processes.

2. **PegasusBase attribute definition (A)** is the library of attributes, i.e., the location where the attribute set A is stored. When a process adds a new attribute, say A_j , to PegasusBase and A_j is not in A , A_j must be added to the library and then any record in PegasusBase can have A_j as its attribute.
3. **Record information (R)** is an array which holds information about the record set R in PegasusBase. Every record has an entry or an index of the array in this model, and every entry has a pointer which points to the first element of the record attribute definition list in the record attribute definition model ($R_k A$). A process can access

a record only from this model. This model acts as a record manager.

4. **Record attribute definition (R_kA)** contains a group of cells or nodes. Each cell can hold up to `MAX_ATT_IN_NODE` attributes. `MAX_ATT_IN_NODE` is a constant that can be defined as any integer. The value used in PegasusBase is 10. In this model, the attributes are described by attribute identification listed in the attribute library. Some of the nodes may link together to form a list of attribute definitions. Each record has one and only one such linked list whose head is stored in the record information model, and the list may become longer as R_kA extends.
5. **The attribute value ($\&R_kA_i$)** is a model that holds the actual value of a record attribute.

5.6.2.1 Shared Memory Information (ShmInf)

As already stated, a region of shared memory is a segment of physical memory which is attached to more than one process's virtual address. The communication among the processes can be set up directly by their shared memory. Different processes have different virtual memories and the address of the shared memory in one process has no meaning to other processes. Pointer values cannot be used as accessing messages

by different processes. The goal of ShmInf is to let multiple processes share the data in shared memory.

Table 5.5 The structure definition of *shm_inf*.

NAME	TYPE	DESCRIPTION
r_inf_offset	integer	It is the offset of the record information block from the beginning of the memory.
r_inf_end	integer	This is the index of the last record in the database.
r_inf_state	state	It is the state of the record information area.
r_att_offset	integer	This is the offset of the record attribute description block.
r_att_end	integer	It shows the index of the last used data attribute.
r_att_state	state	It is the state of the record attribute area.
d_start_offset	integer	This is the offset of record value area.
d_end	integer	It is the last position in the record value area.
d_state	state	It shows the state of the record area.
att_total	integer	This is the total number of attributes used in PegasusBase.
att_start_offset	integer	This is the address of the attribute area. This address is at the bottom of the shared memory and will grow in the reverse direction.
att_state	state	It is the state of the attribute area.

When a region of shared memory is created or attached, the process gets the address of the region, let us call it '*shmadd*'. *Shmadd* is a virtual address and makes no sense in any other processes. Each process attaching the memory has its own '*shmadd*' which may not have the same value as the others.

After the shared memory is created, it is divided into five models as shown in Figure 5.8. All of the models, except *ShmInf*, are empty at the initial state. As PegasusBase grows, the models change from time to time. All of the information describing the models is stored in *ShmInf*. Every process accessing the memory reads the *ShmInf* first and then writes or reads the data in the shared memory.

The way to save the information about the shared memory is by using the offsets of the address '*shmadd*' instead of the addresses in one process. The structure in Table 5.5 is used by the model *ShmInf* and Figure 5.9 shows how the information defines the state of the shared memory.

5.6.2.2 PegasusBase Attributes *A*

PegasusBase attributes *A* is a library that contains all of the attributes used in PegasusBase and their definitions. The size of *A* is initially zero and grows as more attributes are added to the system. The model begins from the bottom (with the highest

Figure 5.9 PegasusBase in shared memory.

address) of the shared memory and extends in the reverse direction, from higher to lower.

Each attribute $A_i \in A$ ($i = 1, 2, 3, \dots, M$) in the model is described by structure *att_dfn* shown in Table 5.6, which contains the *att_name* and *att_size*, the name and size of the attribute, respectively.

Table 5.6 The definition of *att_dfn*.

NAME	TYPE	DESCRIPTION
att_name	string	This is the attribute name.
att_size	integer	It is the attribute size.

Index i of attribute A_i is also the entry of its definition in the block. To define a record is simply to define a set of attribute indices used in that record. This will be discussed in the next section, Record Information.

5.6.2.3 Record Information R

Record information is a table of records. Each record has an index k called the record-ID, where k is the entry of the record in the record information table. This means that each record in PegasusBase R_k , $k = 0, 1, 2, \dots$, has a unique entry k in the table and can be accessed directly according to k . This structure makes the time to retrieve a given record a constant $O(c)$.

The record information model is structured as shown in Table 5.7.

Table 5.7 The structure of *record_inf*.

NAME	TYPE	DESCRIPTION
<i>d_state</i>	state	It is the state of the record.
<i>No_r_att</i>	integer	This is the number of attributes used by the record.
<i>r_att_head</i>	integer	It is the index of the first <i>r_att_node</i> of the data attribute linked list of the data.
<i>r_att_tail</i>	integer	This is the last node of the attribute link list.

In this structure, *No_r_att* is the total number of attributes used in the record; *r_att_head* is a pointer pointing to the head of the record attribute description linked list (see the following sub-topic) and *r_att_tail* points to the tail of the list. A record R_i with 24 attributes and record-ID 5 is shown in Figure 5.10.

5.6.2.4 Record Attribute Definition $R_k.A$

The record attribute definition model is in charge of the particular description of a record. It can be accessed only through the record information model. The organization of mode $R_k.A$ is illustrated in Figure 5.11.

The record information model consists a group of record attribute nodes named *r_att_node*. Each node is defined as in Table 5.8. It includes 10 record attribute descriptors, called *rcd_att*, one empty indicator, called *empty_r_att*, to identify the first empty *rcd_att*, and two pointers, *prevs* and *next*, to provide links to other record attribute

Figure 5.10 The definition of record R_5 in PegasusBase.

Figure 5.11 The organization of record attribute definition.

nodes in the double-linked list.

Table 5.8 The definition of structure *r_att_node*.

NAME	TYPE	DESCRIPTION
red_att[10]	array of record_att	one node contains 10 attributes
empty_r_att	integer	first empty record_att in the node
prevs	integer	index of the previous data attribute node
next	integer	index of the next data attribute node

Table 5.9 The structure of *record_att*.

NAME	TYPE	DESCRIPTION
att_id	integer	index of the attribute in the attribute area
location	integer	index of the address where the value of the attribute is stored; size of the attribute can be obtained from the attribute area
r_att_state	state	state of the record attribute those value can be EMPTY_ATT if the attribute is empty or USEFUL_ATT if the attribute is useful
set_time	integer	time stamp of $\&R_kA_i$

The structure *record_att* in the node is used to record the attribute description. It defines the attribute R_kA_i by a variable *att_id*, keeps the address of $\&R_kA_i$ in *location*, gives the state of R_kA_i at *r_att_state* and stores the time stamp in *set_time*. Table 5.9 shows the definition of structure *record_att*.

With this structure used in the record attribute definition model, each R_kA_i has a

unique definition and a record can have one or several attributes as required. This makes record structure extension easy and convenient.

5.6.2.5 Attribute Values ($\&R_k.A_i$)

Attribute values are stored in an area called *Attribute Values*. The space needed for $\&R_k.A_i$ is determined by the definition of A_i in A and the address of the memory is stored in the record attribute definition model $R_k.A$ in the variable *location* of attribute $R_k.A_i$.

5.6.3 PegasusBase Manager

So far, we have introduced the system data structure and the five models. This section describes how the structure is managed, run-time schema extensions implemented and data retrieved.

Each record, like the example in Figure 5.12, has an entry k ($k = 2$ in the example) in the record information model R . After a record extends its structure from empty to containing some attributes, it may link one or more record attribute nodes, r_att_node , in the record attribute model $R_k.A$ depending on how many attributes are defined to form its structure definition. The value of a record attribute is stored in the attribute value model, $\&R_k.A_i$; the space allocated to the record may not be concatenated. Each attribute $R_k.A_i$ has its own space unit depending on the size of A_i defined in A .

Figure 5.12 A record in PegasusBase.

Since PegasusBase is a class, all items are private unless they are declared after the keyword *public*. A private member cannot be accessed by any function which is not a member of the class. This means that access to certain items of data may be tightly controlled by keeping them private. The system models in PegasusBase are private items; they can only be accessed through public functions. PegasusBase manager is a set of public functions or methods which manage the attributes, records and their values, and control the access to the data in PegasusBase. PegasusBase manager offers many types of services, and system models are well protected and controlled. The services that PegasusBase manager can offer so far includes: *get()*, *set()*, *att_id()*, *att_size()*, *new_att()*, *new_record()*, *version_of()*, *pre_version()*, *clear_record()* and *total_records()* as listed in the Table 5.10. These public functions will be described in the following sections.

5.6.3.1 Define a New Attribute

When created by a process or a knowledge source, PegasusBase is empty, i.e. $A = \{\}$. Any attribute A_i used in PegasusBase has to be defined in A before it can be used. An attribute is defined by adding a new attribute to the attribute library A . The operation *new_att()* is used for this purpose. *New_att()* checks the attribute library, adds the attribute into A if the attribute is not already present and assigns an attribute-ID to the newly added attribute (see Figure 5 (3)). After an attribute is added, it can be used by any record in PegasusBase.

Table 5.10 Public interface.

<code>new_att()</code>	add a new attribute to the system
<code>new_record()</code>	return the first available record id to the user if there are enough empty records available
<code>set()</code>	set a value to an attribute of a record in the PegasusBase
<code>get()</code>	get an attribute value of a record from the PegasusBase
<code>clear_record()</code>	set all of the attributes of a specified record into EMPTY_ATT. It means the value of the record is useless.
<code>version_of()</code>	creates a new version of a given record
<code>pre_version()</code>	returns the parent version of a given record
<code>att_id()</code>	returns the attribute id to the user
<code>att_size()</code>	returns the attribute size to the user
<code>total_records()</code>	returns the total number of records used in the base

New_att() puts new attribute into the library
and assigns an id = 9 to it.

Figure 5.13 Inserting a new attribute into the attribute library.

5.6.3.2 Allocate a New Record

When a new record is required, the operation *new_record()* is used. *New_record()* allocates an unused record in the record information model *R* and returns the record-ID to the user. For example, if R_i is allocated, then the index *i* is returned to the user as the record-ID. A newly allocated record is empty. This is shown in Figure 5.14

Figure 5.14 Allocate a new record.

5.6.3.3 Set a Value to a Record

An attribute does not belong to a record until a value of the attribute is set to the record. This is a very different way to organize attributes and records from that used in conventional systems. A record can own as many attributes as are contained in the

attribute library, and the attribute library can be extended as much as required and as long as the physical memory is available. This architecture makes the system very flexible and efficient. The operation *set()* is responsible for:

- adding an attribute to a record if the attribute does not belong to the record;
- setting a value to an attribute in a record.

As shown in Figure 5.12, a record in PegasusBase has four parts corresponding to the four models, record information R , record attribute definition $R_k.A$, attribute value $\&R_k.A_i$ and attribute description A . Each record has one and only one entry in the record information block. Each entry may have one or more record attribute definition nodes. Every node holds ten attribute definitions which include the attribute-ID and the locations of the values. The algorithm of *set()* is given below:

INPUT:

- record-ID: k
- attribute-ID: i
- attribute value: $value$

RESULT: set the attribute value to the record. If the attribute is not defined in the record, add the definition, allocate memory and store the value in memory.

```

set(k, i, value)
begin
    if k or i is out of range
        return with failure; (1)
    Attsize = att_size(Ai);    {get attribute size} (2)
    if i is not in RkA (3)
        add i to RkA and allocate memory
        with size Attsize;
    store the value at the address; (4)
end;

```

Let's go back to the model *ShmInf*. There are two variables called *r_inf_end*, the end of used record information model *R*, and *att_total*, the total attributes defined in the attribute library *A*. To check if *k* or *i* is out of range involves a simple comparison of *k* with *r_inf_end* and *i* with *att_total*. The first statement is executed in constant time. The second one *Attsize = att_size(i)* is also executed in constant time. In function *att_size()* there is only one statement: *size = Attributes[-i].att_size*, where variable *Attributes* is a pointer member of class *PegasusBase*. It points to the highest address of the shared memory which is used as the *PegasusBase* attribute description library and extends toward the lower address (refer to Figure 5.8). Since pointers can be used in place of array indexing, the time to get the attribute size is constant as well.

The third statement can be separate into three parts: check attribute *A_i*, add *A_i*, and request memory for *A_i*.

- Check A_i :

Let record R_k have n attributes, $n = |R_k.A|$. The simplest way to check if $A_i \in R_k.A$ is to compare each element in set $R_k.A$. The time complexity of the worst case by using sequential search is to compare every element in $R_k.A$ and this has complexity $O(n)$, where n is the number of attributes in the record. Generally speaking, a record does not contain a large number of attributes.

- Add A_i to $R_k.A$:

In the record information model, we defined a variable *r_att_tail* which indicates where the last attribute definition of the record is stored. This means we have a direct accessing mode to the location where the new definition should be added. This step takes constant time $O(C)$.

- Request memory to store the value:

Let's go back to the shared memory information block, *ShmInf*. There is a variable, *d_end*, which records the last allocated space. We can take the required space for the attribute, A_i , in record R_k right away. It is also performed in constant time $O(C')$.

To sum up the three results above, the time complexity of the third statement is $O(n)$, where n is the number of attributes in the record. Therefore the time complexity of the algorithm *set()* is $O(n)$, where $n = |R_k.A|$.

5.6.4 Get an Attribute Value from a Record

The *set()* operation inputs a value into PegasusBase and the *get()* operation outputs a value from PegasusBase. Since the algorithm of *get()* is almost the same as *set()*, except that *get()* reads from the attribute value address while *set()* writes to the address, we do not need to describe it in detail. The time complexity of the *get()* operation is also $O(n)$, where $n = |R_k.A|$.

5.6.5 Clear a Record

When many knowledge sources share the same record, it is not necessary for every knowledge source to know every attribute of the record. For example, to diagnose a computer device, a software expert system knows only those attributes concerning software, while a hardware expert system knows only those concerning hardware. When a knowledge source wants to post new values to a record without confusing other knowledge sources, the value of the record should be cleared before the new values are set. Method *clear_record()* is responsible for this purpose. *Clear_record()* does not wipe out the values of a record, but simply sets each attribute state in the record to *EMPTY* and when we set a new value to the attribute, the state is set to *USEFUL*. After the new values are set, the valid values are only those attributes with state *USEFUL* in the record.

5.6.6 Create a New Version of an Existing Data Structure

In order to change a data structure without damaging it and without altering the previous values of the record, a new data structure version of the record is created by calling method *version_of()* through the class PegasusBase. *Version_of()* is a function deriving a new version of a data structure from a current version. It does two things:

1. apply for a new record header R_{new} ;
2. change the state of R_{new} .

The first one is straightforward, while the second one needs to be discussed in detail.

It can be seen from Figure 5.10 that each record in the record information area R has an entry which includes an integer variable called d_state . This variable has an initial value of MAX_RECORD , the upper bound of record numbers in PegasusBase. Since the record index or ID lies in the range between zero and $MAX_RECORD - 1$, no record can have an ID equal to MAX_RECORD . $d_state = MAX_RECORD$ means the record is a regular record. When a record R_j is derived from R_i , the states of the two records are changed. For R_i , the root record, has $d_state = i$, and for R_j , $d_state = MAX_RECORD + i + 1$. The three states have different meanings:

- $d_state = MAX_RECORD$: a regular record;
- $d_state < MAX_RECORD$: a root record of a different schema version;
- $d_state > MAX_RECORD$: a derived version from record $R_{d_state - MAX_RECORD - 1}$.

Figure 5.15 An Example of Different Versions.

Figure 5.15 presents an example with $MAX_RECORD = 10$, root record R_2 , and four derived records, R_3, R_6, R_7, R_9 which are derived from R_2, R_3, R_6, R_3 , respectively.

5.6.7 Get the Parent Version

In record information model R , we have the version-of relationship information. To get this information, method *pre_version()* is called. *Pre_version()* returns the father version of a given record if there is one, or returns -1 to show that the record has no father version according to the value of *d_state* in the record.

5.6.8 Overloaded Constructors

As described earlier in this chapter, PegasusBase has two parts: current database and history database. The current database is a public database shared among knowledge sources, while history database is a local database used by unique knowledge source only. PegasusBase is defined as a class. Any variable declared as the type of PegasusBase is an object of the class. But how can the two parts be separated? In object-oriented design, this can be done by using the mechanism of constructor overloading, i.e., *polymorphism*. To achieve polymorphism, two constructors (overloaded constructors) with different arguments are defined to initialize PegasusBase, one is for current (public) database, the other is for history (local) database. In the following, we have the constructor headers: one taking no argument and the other taking a character argument. Depending on which constructor is called, the corresponding database will be initialized. Figure 5.16 shows how the overloaded constructors work.

```
PegasusBase();           // Public database initialization
PegasusBase(char old_page); // History database initialization
```

When a process creates an object of PegasusBase as a current database, the current database constructor will be specified. The constructor will create or attach the shared memory depending on whether the shared memory is already created or not. If the process is the one that creates the shared memory, the constructor will lock the entire

Figure 5.16 Overloaded constructors in PegasusBase.

shared memory by calling *intl_lock* in an object of *semaphores*, and will initialize the shared memory. If the process is the one that asks for the shared memory after the shared memory is created, the constructor attaches the memory to the process and gets the current database information from the shared memory information model. The following lists the constructor for the current database in shared memory.

```

PegasusBase::PegasusBase()
{
    shm_state = notok; //the state is not ok to create again
    type = 's';        // set the type as shared memory
    ask_semph intl_sem; // initialize the semaphores operations
    semid = semph.get_semid();
  
```

```

        initialize semaphores
ShmInf = intl_shm();
        // make the shared memory ready to use
char *shmdr = (char*)ShmInf;
        // shared memory address
RecordInf = (record_inf*)(shmdr + ShmInf->r_int_offset);
        // assign the address to the data
        // information pointer
RecordAtt = (r_att_node*)(shmdr + ShmInf->r_att_offset);
        // the address of the data attribute area
DataStart = shmdr + ShmInf->d_start_offset;
        // set the address to the data area
Attributs = (att_dfn*)(shmdr + ShmInf->att_start_offset);
        // set the attribute area
};

```

The main difference of the two constructors is that one creates an entire shared memory with corresponding semaphores and the other simply sets some necessary states for the history database. In history database, the space is allocated when the first action, load history data from a disk file, is required. The list below is the constructor for the history database.

```

PegasusBase::PegasusBase(char old_page)
{ type = 'l';          // Set the memory as local one.
  ShmInf = NULL;        // No space is allocated at initial time.
};

```

5.6.9 Destructor in PegasusBase

A destructor is a member function with the same name as the class plus a leading tilde (~). A class can have only one destructor function which has no argument and no return type. A destructor performs the opposite function of a constructor, that is, cleaning up after an object is no longer needed.

Because there are two types of objects of PegasusBase, the current database in shared memo. and the history database in local memory, the destructor of PegasusBase has two types of cleaning jobs. One is to detach the process from the shared memory or return the shared memory to the operating system depending on whether the process is the last one to detach the memory or not and the other one is to clean up the allocated memory if there exists such a space allocated to the process.

In class PegasusBase, there is a member variable called *type* which distinguishes between objects in the current database and the history database. If *type* = 'l', the database is local (history), if *type* = 's', then the database is shareable (current).

One and only one current database object may be created. In order to control this, a static member variable *shm_state* is used. When *shm_state* is set, no other shared database object can be declared. The destructor can choose the cleaning up method according to the *type* in the class.

When an object is a current database, the destructor checks the number of processes

attached to the shared memory by calling *shmctl()* in UNIX System V. If there are more than one processes attached to the shared memory, the destructor simply unmaps the process, otherwise, the entire shared memory will be returned to the system.

Chapter 6

Data Structure Extension and Versioning in PegasusBase

Having introduced the system design of PegasusBase, this chapter will show how PegasusBase implements data structure extension and versioning, the important data-modeling requirements in the Pegasus project, by using a simple example, “*Carleton*” student database.

6.1 Build the Sample Database

In this chapter, a sample database called “*Carleton*” is used to describe how a PegasusBase is created and extended.

Carleton is a PegasusBase, in which the information about students, courses and grades are stored. Assume that each student in *Carleton* database has two attributes: *st_name* and *st_number*, and the students can choose from courses: *course_A*, *course_B*, *course_C*, *course_D*, *course_B2* and *course_C2*.

To build up this database, we declare *Carleton* as a PegasusBase using the following statement:

```
PegasusBase Carleton();
```

After this declaration, *Carleton* becomes an object of class *PegasusBase*. The constructor of class *PegasusBase* is called through the declaration; it builds up the shared memory and the system models for object *Carleton*. The models A , R , $R_k A$ and $\&R_k A_i$ are empty so far.

6.2 Extend the Data Structure

Two steps are required to extend a data structure:

1. add a new attribute A_j (A_j is not in A) to the *PegasusBase* attribute library A .
2. set a value of the attribute to the record.

Step 1 makes A_j available for the records in *PegasusBase*, but A_j has not been defined in any record yet at this step. Only after the second step is the new attribute A_j set to a record data structure.

Step 2 can be applied to any record in *PegasusBase*. The first value of the attribute set to the record adds the attribute to the data structure.

6.2.1 Define New Attributes in A

In our sample database *Carleton*, there are 8 attributes to be defined in model A , which are: *st_name*, *st_number*, *course_A*, *course_B*, *course_C*, *course_D*, *course_B2* and *course_C2*.

Course_A to *course_C2* are the attributes with user-defined types shown in Table 6.1.

In model A, only the attribute name and attribute size are kept, even if the attribute is a simple type, like integer, character, or a user-defined type, like *course_A* to *course_C2* as in our example.

Table 6.1 The definition of *course*.

c_name	Course name
c_number	Course number
project	Course project
midterm	Mid-term
final	Final

To add these attributes to A the following statements are used:

```
st_name = Carleton.new_att("st_name",20);
st_number = Carleton.new_att("st_number",5);
course_A = Carleton.new_att("course_A",sizeof(course_A));
course_B = Carleton.new_att("course_B",sizeof(course_B));
course_C = Carleton.new_att("course_C",sizeof(course_C));
course_D = Carleton.new_att("course_D",sizeof(course_D));
course_B2 = Carleton.new_att("course_C2",sizeof(course_B2));
course_C2 = Carleton.new_att("course_C2",sizeof(course_C2));
```

The first parameter in *new_att()* is the name of the attribute defined in A; the second one is the size of the attribute. The method returns the attribute-ID which is assigned to

the attribute in A . After these statements are executed, we have: $A = \{ "st_name", "st_number", "course_A", "course_B", "course_C", "course_D", "course_B2", "course_C2" \}$ and $A_1 = st_name, A_2 = st_number, A_3 = course_A, A_4 = course_B, A_5 = course_C, A_6 = course_D, A_7 = course_B2, A_8 = course_C2$.

To share a new defined attribute, a process should know how this attribute is defined. This means there is a protocol among those processes sharing the same attributes. For instans, process C needs to access *course_B*, then process C must know how *course_B* is defined in detail.

6.2.2 Set Values to Records

Assume Suzy and John have registerec as full-time students and have been given their student numbers 12345 and 45678, respectively. To input this information we need to:

1. apply for new records;
2. set the values.

Apply for new records: Any record in PegasusBase has an entry or an index in model R . In inserting a new record into PegasusBase we first need to obtain the record index (or ID). Function *New_record()* is responsible for assigning the index to a new record as its record-ID and registering the associated information. The following

statements give Suzy and John record-IDs:

```
suzy = Carleton.new_record(1);
john = Carleton.new_record(1);
```

The parameter “1” in the above statements means that a single record is requested. If $n, (n > 1)$, multiple records are requested, but only the first index is returned and the other $n-1$ records have indices which follow in succession. In our example, $R = \{R_0, R_1\}$.

Set the values: The records do not have any attribute until some attribute values are assigned to them. The data structure of a record in PegasusBase is defined after an attribute value is set to it, and extended as more values of different attributes are set. The following statements finish both definition of the data structure and assignment of the values to record R_0 , Suzy, and R_1 , John.

```
Carleton.set(suzy, st_name, "Suzy");
Carleton.set(suzy, st_number, "12345");
Carleton.set(john, st_name, "John");
Carleton.set(john, st_number, "45678");
```

Figure 6.1 shows the following results.

$$R_0.A = \{A_1, A_2\}, \quad \&R_0A_1 = 'Suzy', \quad \&R_0A_2 = '12345',$$

$$R_1.A = \{A_1, A_2\}, \quad \&R_1A_1 = 'David', \quad \&R_1A_2 = '45678'$$

Figure 6.1 The result after `set()`.

6.3 Versioning in PegasusBase

Version control is another crucial requirement in the Pegasus project. After a record or an object is created, new versions of the record may be derived from it, more versions may in turn be derived from the existing versions, and so on. Versions provide record evolution states, validity times and information about the alternatives. Figure 6.2 illustrates these relationships for a record *student*. In this example, record *student* has 4 versions — version 2 and 3 are derived from version 1, and version 4 is derived from version 3.

Once the first version has been created, any number of versions may be recursively

Figure 6.2 Version relationships.

derived from it and any subsequent versions. Now, let us consider Suzy chose *course_A* from professor A, *course_B* from professor B and *course_C* from professor C. Three versions, version-2, version-3 and version-4, are created; each professor adds his own course name and the grade list to record Suzy (see Figure 6.3). In the second term, Suzy chose *course_B2* from professor B and *course_C2* from professor C; two new versions, version-5 and version-6, are derived from version-4 and version-3 respectively. New courses are added to different versions.

There are two steps to derive a new schema version from an existing one in

Figure 6.3 Versions of a student record.

PegasusBase.

1. Create the new version from the old version;
2. Add new attributes.

Function “*version_of*” is used in step 1. A new version can keep all or only some of the attributes of the old version. Since an attribute can be added to a schema or data structure at any time, the method, *version_of*, needs to know only which one is the parent version of the new version.

In step 2, the new attributes are added to the new version by the method described in the previous section. The following statements create version-2 to version-6.

```

suzy_2=Carleton.version_of(suzy);           (1)
Carleton.set(suzy_2, course_A, p_course_A); (2)
suzy_3=Carleton.version_of(suzy);           (3)
Carleton.set(suzy_2, course_B, p_course_B); (4)
suzy_4=Carleton.version_of(suzy);           (5)
Carleton.set(suzy_4, course_C, p_course_C); (6)
suzy_5=Carleton.version_of(suzy_4);         (7)
Carleton.set(suzy_5, course_C2, p_course_C2); (8)
suzy_6=Carleton.version_of(suzy_3);         (9)
Carleton.set(suzy_2, course_C2, p_course_C2); (10)

```

Let us look at the first two statements:

1. `suzy_2=Carleton.version_of(suzy),`
2. `Carleton.set(suzy_2, course_A, p_course_A).`

First, *suzy_2* is assigned a record-ID by calling *version_of()* through object *Carleton*.

One parameter, *suzy*, is passed to *version_of()*. Here *suzy* is the record-ID that *suzy_2* is derived from and is set to the *d_state* of *suzy_2* as the parent version automatically.

In the second statement *set()*, we have three parameters: *suzy_2*, *course_A* and *p_course_A*, where *suzy_2* is the record-ID that the new value should be set to, *course_A* is an attribute-ID and *p_course_A* is a pointer that points to the value to be set to the record attribute. This can be represented as:

$$\&R_{suzy-2.A_{course-A}} = *p_course_A.$$

In addition to *version_of()*, there is another function called *pre_version()* which returns the parent version of record-ID. In our example, if *pre_version()* is called as shown below, the result is that: *f_version_of_suzy_2 = suzy*.

```
f_version_of_suzy_2 = Carleton.pre_version(suzy_2)
```

6.4 Retrieving Information from PegasusBase

For a given attribute A_I , ($A_I \in A$), there may be more than one record having a value of $\&R_k A_I$. To find all of the values of A_I in different records, the entire database can be searched to find the record which has attribute A_I by using function *get()*.

For example, to find all of the values of *course_B* in the sample PegasusBase, *Carleton*, the following loop is used in C:

```
int    record_id, i=0;
char   result;
find_struct  find_course_B[20];
ask_semaphore  asem(READ_ONLY);
for (record_id=0; record_id<carleton.total_record(); record_id++)
    { result = Carleton.get(record_id, course_B,
                           &find_course_B[i].value);
      if (result == TRUE)
        { find_course_B[i].record_id = record_id;
          i++; };
```


17

There are two assumptions in this loop:

1. there are no more than 20 records having attribute A_1 ;
2. the structure *find_struct* has been predefined as in Table 6.2.

Table 6.2 The structure of *find_struct*.

NAME	TYPE	DESCRIPTION
record_id	integer	save the record ID here
value	the same structure as the attribute, <i>course</i> in the example shown in Table 6.1.	save the value here

After this loop finishes executing, the values and their corresponding record-IDs are stored in the array *find_course_B*.

All values of attribute A_1 can be found in PegasusBase using this loop. However, there may be some records which have different versions and the versions may have an attribute in common. This means that the values found in this manner may belong to different versions of the same record. To check this, *pre_version()* can be used to find the root record. Those records having the same root record are the different versions of

the root record.

6.5 A Simple Example in Pegasus

A sample student database, *Carleton*, was used to demonstrate how PegasusBase works. This section will illustrate how this system works in Pegasus by using another example.

Assume that a network node with the following attributes is defined:

- **node_name:** name of the current node;
- **node_type:** type of the node, which can be a router, bridge, line or workstation;
- **node_location:** location of the node;
- **node_state:** state of the node, which can be OK, BROKEN or MAINTENANCE;
- **node_connects_to_XXXXX:** where XXXXX is another node name connecting to the current node and the value is the record-ID of node "XXXXX";
- **router:** description of a router;
- **bridge:** description of a bridge;
- **line:** description of a line;
- **workstation:** description of a workstation.

The attribute *node_type* is used to describe the type of the node; this attribute can

**PM-1 3½"x4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT**

either be a *router*, *bridge*, *line* or *workstation*. Table 6.3 shows an example of using the attributes described above in the Pegasus system.

Table 6.3 A simple example in Pegasus.

ATTRIBUTE	RECORD 0	RECORD 1	RECORD 2	RECORD 3
node_name	BCARR1	BMTLB7	ATORL1	BCARW7
node_type	router	bridge	line	workstation
node_location	Ottawa	Montreal	Toronto	Ottawa
node_state	OK	BROKEN	OK	maintenance
node_connects_to _BCARR1		0	0	0
node_connects_to _BMTLB7	1		1	
node_connects_to _ATORL1	2	2		
node_connects_to _BCARW7	3			
router	description			
bridge		description		
line			description	
workstation				description

According to the table, there are twelve attributes, $A = \{node_name, node_type, node_location, node_state, node_connects_to_BCARR1, node_connects_to_BMTLB7,$

node_connects_to_ATOML1, *node_connects_to_BCARW7*, *router*, *bridge*, *line*, *workstation*}, four records, $R = \{R_0, R_1, R_2, R_3\}$, in the database. The attributes in the four records are $R_0A = \{A_1, A_2, A_3, A_4, A_6, A_7, A_8, A_9\}$, $R_1A = \{A_1, A_2, A_3, A_4, A_5, A_7, A_{10}\}$, $R_2A = \{A_1, A_2, A_3, A_4, A_5, A_6, A_{11}\}$, $R_3A = \{A_1, A_2, A_3, A_4, A_5, A_{12}\}$. Where $A_1 = \text{node_name}$, $A_2 = \text{node_type}$, $A_3 = \text{node_location}$, $A_4 = \text{node_state}$, $A_5 = \text{node_connects_to_BCARR1}$, $A_6 = \text{node_connects_to_BMTLB7}$, $A_7 = \text{node_connects_to_ATOML1}$, $A_8 = \text{node_connects_to_BCARW7}$, $A_9 = \text{router}$, $A_{10} = \text{bridge}$, $A_{11} = \text{line}$, $A_{12} = \text{workstation}$.

Let us call the database *simple* and assume that *simple* has already been initialized and the appropriate information has been entered. The database is now going to be used to:

1. find out which device or line is broken;
2. add a new node type, *Brouter*;
3. change the *node_type* from *router* to *Brouter* in record 0 (create a new version of record 0);

6.5.1 Find out What Is Broken

To find out what is broken, it is necessary to find which record has the attribute

node_state and then check the value. This can be done by changing the loop in Figure 6.4 as follows:

```
int    record_id, i=0, find_state;
char   result;
find_struct  find_broken[20];
ask_semph  asem(READ_ONLY);
for (record_id=0; record_id<simple.total_record(); record_id++)
    { result = simple.get(record_id, node_state, &find_state);
      if (result == TRUE && find_state == BROKEN)
          { find_broken[i].record_id = record_id;
            simple.get(record_id, node_name, &find_broken[i].name);
            i++; }
    };
```

The meaning of *find_struct* is slightly different from the one used in Table 6.2.

Here *find_struct* has a *name* instead of a *value*.

Upon completion of the above loop, the *node_names* and record-IDs with *node_state* will be stored in the array *find_broken*.

6.5.2 Add a New Node Type

Assume there is a new node type, called *Brouter*, which is to be added to the database *simple*. *Brouter* has its own data structure, called *Brouter_struct*, to describe its features. The following statement is used to add *Brouter* into PegasusBase *simple*.

```
Brouter = simple.new_att("Brouter", sizeof(Brouter_struct));
```

Simple.new_att() returns the attribute-ID of variable *Brouter*.

6.5.3 Create a New Version

Node *BCARR1* is upgraded from type *router* to *Brouter*. A new version of the record must be created and the old version must be kept for diagnosis. The new version is created by calling the function *version_off()* which sets the appropriate values in the new version. In the example, the following statements are necessary.

```
bcarr1_2=simple.version_of(bcarr1);
simple.set(bcarr1_2, node_name, "BCARR1");
simple.set(bcarr1_2, node_type, "Brouter");
simple.set(bcarr1_2, node_location, "Ottawa");
simple.set(bcarr1_2, Brouter, the address of the description);
simple.set(bcarr1_2, node_connects_to_BMTLB7, 1);
simple.set(bcarr1_2, node_connects_to_ATOML1, 2);
simple.set(bcarr1_2, node_connects_to_BCARW7, 3);
```

Chapter 7

Conclusions

7.1 Contributions

PegasusBase is a run-time data structure extension system, a real time data manager and a multiple process/user communication tool. This combination of features is, as far as is known, unique. All of the features described in this thesis have been implemented and are operational. The system was tested on 10 million messages, which have 128 bytes in size on average, and four knowledge sources and took 37 ± 1 seconds of CPU time on a sparc 2 workstation. This shows that PegasusBase in shared memory is a very fast database, can meet the system requirements and can run smoothly in the simulation environment.

PegasusBase employs a data model which has an attribute library, $A = (A_1, A_2, \dots, A_l)$, that holds all of the attributes used in the system, a record information set, $R = (R_1, R_2, \dots, R_m)$, which stores all of the records in the system, a record attribute definition group, $R_k A$ ($k = 1, 2, \dots, m$), that describes the record attributes and an attribute value area, $\&R_k A_i$ ($k = 1, 2, \dots, m \quad i = 1, \dots, l$), that stores the values of the attributes. With this model, a new attribute A_{l+1} not in A can be added easily; any attribute $A_i \in A$ may be added to any record R_k by setting a value to $R_k A_i$. The

data structure of record R_k extends as a new attribute A_j is added to $R_k.A$ and the time complexity of adding a new attribute is $O(|R_k.A|)$.

Using shared memory using UNIX System V speeds up the communications among knowledge sources, because the knowledge sources can share the information directly through the common memory attached to each of them.

7.2 The Use of C++

Knowledge sources in Pegasus are written in several different languages, such as C, Pascal and a COSMIC CLIPS shell, which are function-oriented methods. PegasusBase was designed with object-oriented methods, implemented in C++ and is an attempt to mix object-oriented and function-oriented methods in one system. Using polymorphism, inheritance and class constructors/destructors, the features unique to object-oriented methods, PegasusBase demonstrates some advantages. It requires less coding, hides the implementation details, and is easy to extend and test. In addition, it is easy to integrate with other systems regardless of whether they are object-oriented or function-oriented. This is because C++ continues the C tradition and extends C into the object-oriented programming world.

In PegasusBase, polymorphism is used to construct the two related but slightly

different databases, current database and history database. Class PegasusBase inherits the properties of class *semaphores* which exists in current database only. Besides polymorphism and inheritance, destructors, the complement of the constructors, are employed to perform some actions when objects are destroyed. This is particularly useful when the shared memory is accessed by control semaphores. Local objects of *ask_semph* are created when accessing shared memory and there is no need to worry as to when the semaphores should be cleared. When the function finishes, the destructor will be called automatically to clear the semaphores whether the procedure exits with the correct result or an error.

From the experience gained in designing and implementing PegasusBase in C++, it is the author's opinion that the improvement from C to C++ is similar to the improvement from Basic to Pascal. Object-oriented programming does offer a better organization and method for software design and software management.

7.3 Future Work

PegasusBase is a data manager in the Pegasus project and can also be a data model for an object-oriented database. However, an object-oriented data model must capture object-oriented concepts: encapsulation and extensibility. In an object, some data and procedures operating on the data are encapsulated; the data is the state of the object,

and the code is the behavior of the object. In PegasusBase, the extensibility in data structures is already present, but there is still a need to add another level to the methods so that the system can capture object-oriented concepts and become an object-oriented database model.

Retrieval facilities offered by PegasusBase are based on the requirement of the blackboard system used in Pegasus. They are efficient, however they are not self-sufficient if PegasusBase is used without the blackboard system. More strategies for processing a query must be added to PegasusBase so that a user can retrieve information from the database in an easier manner.

With regard to versioning control, PegasusBase offers basic management, but this still has some limitations. If a new level is added to make versioned records fully controlled, the database system will offer more versatile services with regard to versioning control.

References

- [1] B. Pagurek, N. W. Dawes and A. R. Kaye. *A Multiple Paradigm Diagnostic System for Wide Area Communication Networks*. Carleton University Technical Report SCE 91-37, 1991.
- [2] N. W. Dawes, A. R. Kaye, and B. Pagurek. *Pegasus Demonstration*, Oct. 1991. Systems and Computer Engineering, Carleton University.
- [3] S. Iqneibi. *A Blackboard Architecture to Support Network Fault Diagnosis*. Master's thesis, Carleton University, Department of Systems and Computer Engineering, 1992.
- [4] N. W. Dawes. *Dragon: a very high speed, very large scale network simulator*. Telecommunications Research Institute of Ontario, Technical Report SCE-91-31.
- [5] N. W. Dawes, J. Altoft and B. Pagurek. *Effective Network Diagnosis by Reasoning in Uncertain nested Evidence Spaces*. Carleton University Technical Report SCE 91-26; Avignon' 1992:12th International Conf., pages 403-414, Jul. 1991.
- [6] R. Dussault. *A Diagnostic Expert System for Wide Area Networks*. Master's thesis, Carleton University, Department of Systems and Computer Engineering, 1992.
- [7] F. Viens. *A Neural Network Approach to Detect Traffic Anomalies in a Communication Network*. Master's thesis, Carleton University, Department of Systems and Computer Engineering, 1992.
- [8] J. Embry, P. Manson, D. Milham. *An open Network Management Architecture: OSI/Nm Forum Architecture and Concepts*. IEEE Network Magazine, pages 14-22, July 1990.
- [9] A. Dupuy, S. Sengupta, O. Wolfson and Y. Yemin. *NETMATE: A Network Management Environment*. IEEE Network Magazine, pages 35-43, Mar. 1991.
- [10] C. A. Joseph, K. H. Muralidhar. *Integrated Network Management in an Enterprise Environment*. IEEE Network Magazine. pages 7-13, July 1990.
- [11] Lisa Olson, Alan Blackwell. *Understanding Network Management with OOA*. IEEE Network Magazine, pages 23-28, July 1990.
- [12] AT&T Technical Reference. *AT&T Network Management Protocol Specification — Transport Through Application Layers*, June 1989.

- [13] ISO/IEC 7498-4. *Information Processing System —Open System Interconnection —Basic Reference Model —Part 4: Management Framework*, 1989.
- [14] ISO/IEC JTC 1/SC21. *2nd DP 10040: Information Processing Systems-OSI- Systems Management Overview*, Dec. 1989.
- [15] Fred Halsall, Nasser Modiri. *An Implementation of an OSI Network Management System*. IEEE Network Magazine, pages 44–53, July 1990.
- [16] J. Lightsey Wallace. *Network Management and the Corporate Mission*. In *Integrated Network Management I*, pages 93–100. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1989.
- [17] Zheng Wang. *Model of network faults*. In *Integrated Network Management I*, pages 345–352. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1989.
- [18] J. Goldman, P. Hong, C. Jerommimon, G. Louit, J. Min and P. Sen. *integrated fault management in interconnected networks*. In *Integrated Network Management I*, pages 333–344. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1989.
- [19] C. Joseph, J. Kindrik, K. Muralidhar, C. Co, T. Toth-Fejel. *Map Fault Management Expert System*. In *Integrated Network Management I*, pages 627–636. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1989.
- [20] Technology at Research Triangle Park. *IBM. Additional Improvements in IBM Network Management Interface*, Sep/Oct 1987. 11(2):1 — 12.
- [21] AT&T. *AT&T network Management Protocol Specification*, June 1989.
- [22] Alain Mouttham, Monica Frontini, Jonathan Griffin, Simon Lewis. *LAN Management Using Expert Systems*. In *Integrated Network Management I*, pages 491–502. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1989.
- [23] Ta Kuya Yamahira, Yoshiaki Kiriha, Shiro Sakata. *Unified Fault Management Scheme for Network Troubleshooting Expert System*. In *Integrated Network Management I*, pages 637–646. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1989.
- [24] Bruce L. Hitson. *Knowledge-Based Monitoring and Control: An Approach to Understanding the Behavior of TCP/IP Network Protocols*, 1988.
- [25] Rodney M. Goodman, John Miller, Padhraic Smyth,. *Real Time Autonomous*

- Expert Systems in Network Management*. In *Integrated Network Management I*, pages 599–620. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1989.
- [26] R. M. Goodman and H. Latin. *Automated Knowledge Acquisition from Network Management Databases*. In *Integrated Network Management II*, pages 541–549. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1991.
 - [27] Paul Hong, Prodip Sen. *Incorporation Non-deterministic Reasoning in Managing Heterogeneous Network Faults*. In *Integrated Network Management II*, pages 481–503. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1991.
 - [28] Monica Frontini, Jonathan Griffin, Simon Towers. *A Knowledge-Based System for Fault Localization in Wide Area Networks*. In *Integrated Network Management II*, pages 519–530. Elsevier Science Publishers B.V. (North — Holland) IFIP, 1991.
 - [29] N. W. Dawes. *Pegasus — General Architecture and Designs*. Telecommunications Research Institute of Ontario, Dept. of Systems and Computer Engineering, 1991.
 - [30] Won Kim. *Introduction to Object-Oriented Database*. The MIT Press, 1990.
 - [31] ACM, New York, New York. *CODASYL Data Base Task Group April 71 Report*, 1971.
 - [32] Henry F. Korth, A. Silberschatz. *Database System Concepts*. McGraw-Hill Book Company, 1986.
 - [33] Bipin C. Desai. *An Introduction to Database systems*. West Publishing Company, 1990.
 - [34] Paolo Atzeni, Riccardo Torlone. *Efficient Updates to Independent Schemes in the Weak Instance Model*. ACM SIGMOD RECORD, Volume:19, Issue 2, 1990.
 - [35] Michael J. Carey. *A Data Model and Query Language for EXODUS*. ACM SIGMOD RECORD, Volume 17, No. 3, 1988.
 - [36] R. Langerak. *View Updates in Relational Databases with an Independent Scheme*. ACM Transactions on Database Systems, Volume 15, No. 1, Mar. 1990.
 - [37] Michael J. Carey, David J. DeWitt, Scott L. Vanderberg. *A Data Model and Query Language for EXODUS*. Proc. SIGMOD Record, Volume 17, No. 3, Sept. 1988.
 - [38] L. Rowe, and M. Stonebraker. *"The POSTGRES Data Model"*. Proc. Intl. Conf. on Very Large Data Bases, Brighton, England, Sept. 1987.

- [39] W. Kim et al. *Architecture of the Orion Next-Generation Database System*. IEEE Trans. Knowledge and Data Eng., Volume 2, No. 1, 1990.
- [40] O.DEUX et al. *The Story of O2*. In *IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING*, volume 2 No. 1, March 1990.
- [41] R. Breidl et al. *The Gemstone Data Management System*. In W. Kim, F. Lochovsky, editor, *Object-Oriented Concepts, Databases, and Applications*, pages 283–308. Addison-Wesley, Reading, Mass, 1989.
- [42] Elisa Bertino. *Object-Oriented Database Management Systems: Concepts and Issues*. IEEE Computer, Apr. 1991.
- [43] Katsumi Tanaka, Masatoshi Yoshikawa, Kozo Ishihara. *Schema Virtualization in Object-Oriented Databases*. Proc. IEEE Conf. on Data Engineering, Los Angeles, CA, Feb. 1988.
- [44] Suad Alagic. *Object-oriented Database Programming*. Springer-Verlag, New York, 1988.
- [45] A. Björnerstedt, C. Hulten. *Version Control in an Object-Oriented Architecture*. In W. Kim, F. Lochovsky, editor, *Object-Oriented Concepts, Databases, and Applications*, pages 451–485. Addison-Wesley, Reading, Mass, 1989.
- [46] K. Wilkinson, P. Lyngback, W. Hasan. *The Iris Architecture and Implementation*, volume 2, No. 1. IEEE Trans. Knowledge and Data Eng., 1990.
- [47] Jay Banerjee, Won Kim. *Semantics and Implementation of Schema Evolution in Object-Oriented Databases*. Proc. SIGMOD Conf. San Fran., May 1987.
- [48] William Kent. *An Overview of the Versioning Problem*. Proc. SIGMOD Record, Volume 18, No. 2, Jun. 1989.

END

15-1 2-93

FIN