DATABASES FOR DISTRIBUTED REALTIME SYSTEMS

by

Xenophon Andriopoulos

January 1986

Imperial College
Department of Computing
180 Queensgate
London SW7 2BZ, U.K.
Tel:01-5895111

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering of the University of London, and for the Diploma of Membership of the Imperial College.

ABSTRACT

Applying database concepts to realtime systems can bring significant improvements in the development and maintenance of such systems, especially if the application is itself distributed. This thesis makes three contributions: a detailed survey of the database needs and requirements for realtime systems; a proposal for a new realtime database model; and a component architecture for implementing the model.

Realtime systems require a realtime database for holding the current and past state of the environment and a configuration database for supporting the overall management of the system. A **realtime database** requires fast response times, high reliability and availability, ease of expansion and change, and is often distributed. A **configuration database** contains the descriptions and the configuration of the components of the system and is similar to conventional databases. Existing database models are not really suitable for a realtime database although they can be used for a configuration database. A new and improved realtime database model is therefore proposed in this thesis.

According to the model, a realtime database consists of a number of data modules. A **data module** defines data objects, operations and views on the data objects, and performance characteristics. A view is a collection of data objects and operations exported by a data module, which can subsequently be imported by another. Thus, access is restricted to the data objects and operations specified in the view. The performance characteristics of a data module provide abstract treatment for response times, reliability backups, and levels of consistency. Individual data modules are specified in a **data definition language** (DDL).

Instances of these data modules can then be created and interconnected to form a **database module.** The interconnection binds an imported view of a module instance to the exported view of another instance. Database modules are specified in a **database configuration language** (DCL). A database module can contain instances of other database modules in a nested manner. The DCL can also be extended to specify configuration changes. This separation between the DDL and the DCL facilitates the development of a realtime database and provides the flexibility in adapting to changes. The approach is novel in that it can allow dynamic reconfiguration of the database without shutting down the whole system.

A component architecture, essentially a refinement of the model, is used for the implementation of the model. The architecture provides a few basic components that can be interconnected in order to achieve the overall functionality of a data module. The model has been applied successfully to four case studies including a patient monitoring system for which a prototype implementation has also been developed. The conclusions of these case studies, supported by design experiences and performance measurements acquired from the prototype implementation, indicate the viability and practicality of the proposal.

To my parents

Prelude

The Grin

Of all the places I saw,
of all the words I heard,
of all the poems I wrote,
of all the songs I sang,
of all the dreams I had,
Nothing could make me happy.

It was only
that spontaneous grin,
that strange amalgam
of failure, irony and hope,
that really gave me strength
To face the ghost of happiness.

X.A. 7-10-84

CONTENTS

1. INTRODUCTION 1

1.1 The Problem Domain 1

1.2 The Approach 2

1.3 Design Principles 3

1.4 Thesis Outline 5

2. REQUIREMENTS 6

2.1 The RealTime Environment 6

2.2 The Need for Databases 10

2.3 The RealTime Database 11

2.4 The Configuration Database 19

2.5 Chapter Summary 22

3. RELATED WORK 24

3.1 Network & Hierarchical Models 24

3.2 Relational Model 25

3.3 Semantic Models 27

3.4 ANSI Architecture 28

3.5 NBS Architecture 30

3.6 CERN SPS Model 30

3.7 Global Broadcast Model 31

3.8 CUTLASS Model 32

3.9 Chapter Summary 33

4. REALTIME DATABASE MODEL 34

4.1 Model Overview 34

4.2 The Data Definition Language (DDL) 36

4.3 The Database Configuration Language (DCL) 47

4.4 DCL Extension for Database Modification 54

4.5 Chapter Summary 59

5. A COMPLETE EXAMPLE 60

5.1 Data Types 63

5.2 Data Modules 64

5.3 Database Modules 68

5.4 Modifications 69

5.5 Further Modifications 70

5.6 Chapter Summary 72

6. IMPLEMENTATION 73
6.1 Component Architecture 73
6.2 Performance Characteristics 76
6.3 Run-time Mechanisms 78
6.4 The Conic Environment 89
6.5 Prototype Implementation 90
6.6 Performance Issues 92
6.7 Chapter Summary 94

7. DISCUSSION 95
7.1 Design Influences & Rationale 95
7.2 Comparison with Abstract Data Types 98
7.3 Possible Extensions 99
7.4 Use of the Model 101
7.5 A RealTime System Development Method 102
7.6 Criteria for Decomposition & Allocation 104
7.7 Chapter Summary 106

8. CONCLUSIONS 107
8.1 Summary of Achievements 107
8.2 Critical Evaluation 107
8.3 Design Principles Revisited 109
8.4 Suggestions for Further Work 110
8.5 Final Remarks 113

APPENDIX: Formal Definitions 114

REFERENCES 123

Acknowledgements

Epilogue

CHAPTER 1

INTRODUCTION

Databases have been used in computer systems for commercial applications to separate the application programs from the storage and retrieval of data. Such a separation can lead to very flexible systems with significant benefits upon their development and maintenance. This thesis attempts to extend the use of databases to realtime systems in order to obtain similar benefits. The contributions of the thesis are: a detailed survey of the database needs and requirements for realtime systems; a proposal for a new realtime database model; and a component architecture for implementing this model.

1.1 The Problem Domain

Realtime systems monitor and control an environment by receiving data from sensors, doing some processing, and returning the results via actuators sufficiently quickly to affect the state of the environment. Failure to respond within a predefined time interval may have catastrophic consequences. Response times therefore must be predictable and guaranteed, and very high reliability and availability are required. Realtime systems are used for a variety of applications such as the generation and transmission of electricity, chemical and petrochemical plants, mine automation, robot control, telecommunications, transport control, and patient monitoring in hospitals.

Technological advances during the last decade have allowed distributed microcomputers to be used for the implementation of realtime systems. These **distributed realtime systems** have several advantages over their centralised predecessors: improved response times, higher reliability and availability, inherent modularity and flexibility, thus potential ease of expansion and modification. Many manufacturers compete in the market of distributed realtime systems and offer a variety of products that can be configured from a generic system of hardware and software components. This approach speeds up the development of a realtime

system, but unfortunately all the subsequent modifications are confined to the predefined components of the generic system. As a result, such generic systems tend to be rather inflexible. New hardware and software components cannot be added. User-defined software components are not possible without application programming, except for user-defined display frames. Changes are often difficult to carry out and sometimes require that the whole system be shut down. Clearly, there is a need for better configuration facilities that will fully exploit the inherent flexibility of distributed systems.

1.2 The Approach

Databases have been used for many commercial applications to separate the application programs from the storage and retrieval of data. Such a separation provides the flexibility to expand and modify the functions performed on the data with relative ease. This thesis sets out to extend the use of databases to realtime systems in order to achieve similar flexibility. More precisely, the objective of the thesis is to investigate the database needs of realtime systems and to propose a database model and some tools that may be used to fulfil these needs.

The approach taken was to start with a requirements survey for realtime systems and hence pinpoint their database needs. A thorough examination of existing realtime systems shows that two different databases are required: a realtime database for providing separation and cleaner interfaces between application programs, and a configuration database for supporting the development, subsequent modifications, and overall management of the system. A **realtime database** contains the current and past state of the environment and requires fast response times, high reliability and availability, ease of change and extension, and is often distributed. Sometimes, it is also necessary to provide dynamic reconfiguration, i.e. the ability to introduce changes while the system is running. A **configuration database** contains the descriptions and the configuration of the various components within the system. Its response times are not critical, dynamic reconfiguration is not required, and the database need not be distributed. High reliability and consistency are however important. A configuration database is similar to a software engineering database, thus it represents a more general need related to the management of all computer systems.

Database models for commercial databases (relational, semantic, etc) are not suitable for a realtime database because they fail to address the crucial requirements for response times, reliability, availability, distribution and dynamic reconfiguration. Other database models which have been developed specifically for realtime systems, impose limitations on the data they can represent, and do not always allow dynamic reconfiguration. Thus, a new and improved realtime database model is proposed in this thesis. Some of the existing database models however are suitable for a configuration database. This thesis suggests a semantic model as the best candidate for a configuration database.

The proposed realtime database model provides the necessary concepts and a notation for the logical specification of a realtime database. The model is novel in several ways. It separates the definition of database components from the configuration of a database as interconnected instances of components. This separation facilitates the development of the database and provides the flexibility in adapting to changes. The database is decomposed to a set of components with well-defined interfaces between them. Individual database components are easier to understand, specify, implement and modify. As a result, the entire database is easier to design, specify, implement and maintain. The approach is novel in that it can allow dynamic reconfiguration of the database without shutting down the whole database or system. Views are used to provide interface control between components and nested database configurations are allowed. The model also provides a new kind of abstraction called performance abstraction which allows abstract and formal treatment for the performance characteristics of the database.

The design and validation of the model was based on a series of four case studies. One of the case studies concerns a patient monitoring system for which a prototype implementation has been developed. The implementation is in terms of a component architecture that comprises a few basic components together with the interconnections between them. The conclusions of the case studies, supported by experiences and performance measurements acquired from the prototype implementation, indicate the viability and practicality of the approach.

1.3 Design Principles

Like most other work in computer science, the realtime database model

proposed in this thesis constitutes a synthesis of ideas and experiences which were selected to satisfy the requirements for a realtime database. These ideas and experiences were also systematically unified to obtain the most basic concepts that can be used for the specification of a realtime database. Such a unification however can be successful only if it is guided by some principles that will prevent complex interactions from making the design of the model intractable. The realtime database model has evolved from the following design principles:

* Consistency: This is also known as "conceptual integrity" and demands that the model adheres to a set of coherent concepts [Brooks 75]. Good concepts that do not integrate nicely with the model are best left out. Consistency determines the unity of the design and ultimately the ease of use of the model.

* Simplicity: In this thesis, simplicity is interpreted as minimality of concepts [Brooks 75, Kent 78]. A simpler model is easier to understand, learn and use. It may reduce the likelihood of errors and misuse of its concepts. It will also be easier to implement and will make modest demands for resources. This is particularly desirable since the target environment comprises simple microcomputers with limited resources.

* Utility: The model should not be considered only as an interesting academic endeavour but also as a tool that will be used to design real systems (cf. correspondance between "data" and "reality" in [Kent 78]). Thus, the model should be usable, teachable and applicable to large and complex systems. Utility proceeds from both consistency and simplicity. However, simplicity per se is not valued as an absolute principle [Hilfinger 81], but must be considered relatively to the functions built in the model. Utility is therefore taken to signify a tradeoff between simplicity and the useful functionality of the model.

* Flexibility: This refers to the ease of change and extension of the various concepts in the model. Ease of change requires that the concepts of the model should be loosely coupled: a small change to a concept should have only minor impact on the other concepts. Ease of extension demands that the model is open-ended to accomodate additional concepts, and able to be incorporated in other models.

1.4 Thesis Outline

This thesis is divided into eight chapters and an appendix which are organised as following:

Chapter 2 presents the general requirements for realtime systems and then proceeds to identify the particular requirements for the realtime database and for the configuration database of a realtime system.

Chapter 3 describes the main features of existing database and proposed reference models. These features are critically examined with regard to their relevance and limitations for realtime systems.

Chapter 4 proposes a new and improved realtime database model, i.e. the concepts and a notation for the logical specification of a realtime database. This chapter includes descriptions of a data definition language (DDL), a database configuration language (DCL), and an extension to the DCL for performing database modifications.

Chapter 5 demonstrates the use of the model in real world situations by means of an example, a simplified patient monitoring system.

Chapter 6 presents a component architecture that can be used for the implementation of the model. The architecture comprises a few basic components together with the possible interconnections between them.

Chapter 7 discusses the influences and rationale behind the design of the model and suggests possible extensions and uses for it. A realtime system development method using the model is also outlined together with some criteria for the decomposition and allocation of data.

Chapter 8 concludes with a summary and a critical evaluation of the proposed model and suggests some directions for further work.

The Appendix lists the formal BNF definitions for the DDL, the DCL, and the extension to the DCL for performing database modifications.

CHAPTER 2

REQUIREMENTS

Realtime systems are characterised by a set of stringent requirements which clearly distinguish them from other computer systems. This chapter surveys the general requirements for realtime systems and investigates their need for databases. The need for both a realtime database and a configuration database are identified and the particular requirements for these two different databases are included in this chapter.

2.1 The RealTime Environment

Realtime systems monitor and control an environment which often comprises a plant with one or more (physical) processes. They are used for many industrial applications such as the generation and transmission of electricity, chemical and petrochemical plants, oil refineries, mine automation, high-energy laboratories, steel production, pulp and paper mills, robot control, and the automobile industry. Other applications which become increasingly common, are telecommunications, control and command systems, radar systems, environmental monitoring, transport control, and patient monitoring in hospitals. The requirements that the environment imposes upon realtime systems are now presented in detail.

2.1.1 Sensors & Actuators

The system interacts directly with its environment via special hardware devices: sensors (input points) and actuators (output points). It accepts data from sensors, performs a computation, and uses the results to control actuators. The consequences of a control action are fed back to the system through the outside environment and the sensors. This is called closed loop control. Control decisions can also be made by a human who instructs the computer to perform a control action. This is called open loop control. The system then has to provide man-machine interfaces which allow human users to influence the results of computations as well as to affect the environment directly.

2.1.2 Response Times

The term response time refers to the time interval needed to react to an external stimulus with an appropriate action. Realtime systems require response times as low as a few milliseconds or even microseconds in some cases. Missing a response may result in a disaster, so response times must be predictable and guaranteed. This means that the correctness of a realtime system depends both on its correct functional behaviour and on its response times [Wirth 77]. Such response times requirements are hard to meet and tend to push computer technology to its limits [Hansen 78].

2.1.3 Reliability

Reliability is important because it contributes to both safety and availability. Safety is of crucial importance because of possible injury or even death, destruction of property or equipment, and environmental damage or pollution [Leveson 84]. Some realtime systems are expected to provide continuous operation 24 hours a day, so high availability is necessary to avoid disruption of services or loss in production. A highly available system is expected to have more than 99% availability [Kim 84]. Some existing realtime systems have almost eliminated downtime by achieving much more than 99% overall availability and the mean time between failures that may result in complete loss of the control of the environment is quoted to be 50-100 years [Sandoz 82].

2.1.4 Flexibility

Flexibility refers to ease of extension (extensibility) and to ease of change (modifiability). Extensibility reflects the incremental growth approach often adopted for the introduction of a realtime system. Modifiability is required for several reasons: the environment evolves and undergoes continuous change; faulty components will have to be replaced; new technology will gradually be introduced; the plant may have to be reconfigured to optimise performance; even the introduction of the system itself will act as a stimulus for further change. Flexibility dictates that the design of the system is open-ended and able to adapt to change. Because of the requirement for continuous operation, extensions and changes should be performed online without shutting down the whole system. Hence, **dynamic reconfiguration** is necessary [Magee 84, Schell 71]. Dynamic reconfiguration can also be

used in conjunction with automatic failure detection in order to overcome failures automatically without human intervention [Dowson 84].

2.1.5 Concurrency

There is inherent concurrency between the entities of the environment and in their interaction with the system. This has led many designers to structure their realtime systems as a collection of concurrent tasks (processes or activities) which attempt to cope with the concurrent, asynchronous and non-deterministic occurence of the various events from the environment. A task may be thought of as a sequential program in a conventional programming language. These tasks normally reside in main memory, exhibit time dependencies, and never terminate but loop forever. Concurrency has also led to the attendant requirement for synchronisation between tasks [Wirth 77]. Note that there is a close interaction between the tasks of a realtime system because they cooperate towards a common overall goal. In contrast, office automation, university networks and other resource sharing systems exhibit much looser interactions between their concurrent tasks.

2.1.6 Static Structure

The environment exhibits a rather static structure. Plant equipment is allocated to permanent positions. Sensors and actuators reside at fixed locations. This has made most designers adopt the static structure assumption for their realtime systems [Wirth 77, Hansen 78, Kramer 79, Franta 81, Halling 81, Magee 84]. A system is designed to use a fixed amount of resources and to comprise a fixed number of tasks. The functions of each task are fixed and known in advance. Storage can be allocated statically for each task, provided that neither recursion nor unbounded data structures are used. For distributed systems, a task is permanently allocated to a single physical node. It must be pointed out that the structure of the system remains static at run time, but the flexibility of section 2.1.4 must be provided to modify the structure of the system at reconfiguration time.

Static structure considerably simplifies the design of a realtime system and avoids the problems associated with the dynamic creation and migration of tasks. Static structure also allows for more deterministic system behaviour. Guaranteed maximum response times can be derived for

each task by means of deadline calculations that are based on the time dependencies between tasks [Wirth 77, Kramer 79, Haase 81].

2.1.7 Regularity

Some of the functions in a typical realtime system are executed periodically whereas others are event-driven. The ratio of periodic to event-driven functions depends very much on the particular environment, but the former always constitute a significant proportion [Franta 81]. Under steady-state conditions, this periodicity in conjunction with the static structure of the system results in a remarkable regularity on the execution of the various functions. Regularity simplifies the design of the system because it is easier to allocate periodic functions to tasks [Gomaa 84, Franta 81]. It is a rare situation that all the event-driven functions are triggered simultaneously. This can happen in an emergency situation of plant-wide alarms or failure, which produces a burst of events. Such a burst of events causes a large and immediate increase in the processing and communication load of the system, thus it constitutes the worst-case (upper bounds) for the response times of the system.

2.1.8 Hierarchical Organisation

A multilevel control hierarchy has been widely used as the basis for the organisation of realtime systems [Mesarovic 70, Williams 78, Halling 81]. This is a logical organisation which enhances the understanding and simplifies the design of large and complex systems. A typical hierarchy comprises the company headquarters at the top, then the central control rooms, the units and operations, the direct digital controls, and finally the sensors and actuators. This is reflected in the hardware configuration of some systems. Such a hierarchy is also used to allocate the various tasks to specific levels according to their functions, and to arrange the human users according to their responsibilities and assignments [Williams 78].

2.1.9 Distribution

Advances in computer communications during the last decade resulted in the advent of industrial local area networks which allowed realtime systems to be implemented as distributed systems. A distributed system is attractive for several reasons. It can offer increased performance,

higher reliability, graceful degradation, and inherent flexibility at a relatively low cost. It can reflect the physical distribution of plant equipment and devices over a geographical area. It can also be used to integrate together a number of isolated systems (such as controllers, robots, production management, inventory control, etc) in order to achieve the goals of total plant automation and computer integrated manufacturing. Thus, the stage has been reached where almost all realtime systems are or will be distributed [Halling 81, Jervis 83].

2.2 The Need for Databases

Many manufacturers compete in the market of realtime systems and most of them offer very attractive distributed realtime systems [Blickley 84, Frost 84 & 83]. Their basic product is a generic system of modular components: plant interfaces (input and output points), preprogrammed and programmable controllers, a communication system (often a serial broadcast highway), several man-machine interfaces, and a management system. This generic system can be tailored to the needs of a particular plant through configuration [Sandoz 82, Halling 81]. Configuration is usually carried out in the form of "database definition" [Marsh 79] where an engineer defines the components and parameters of the system by means of data tables, control diagrams and specification charts, but does not have to write any software. This speeds up system development, but unfortunately restricts modifications to the predefined components and limits of the generic system. As a result, the generic systems supplied by the various manufacturers tend to be rather inflexible. New hardware and software components cannot be added. User-defined software components are not possible without application programming, except perhaps for user-defined display frames. Changes are often difficult to carry out and sometimes require that the whole system or plant be shut down. Clearly, there is a need for better configuration facilities that will match the inherent flexibility of distributed systems.

Databases have been used in many commercial applications to isolate the programs from the representation and storage of the data which they access. Often the data is more stable and changes less frequently than the programs. This separation provides the flexibility to accomodate unforeseen access to data, so it becomes easier to expand and modify the programs using shared data. In fact, the database may be seen as the backbone or skeleton that gives shape to the entire system. Similar

benefits can accrue from the use of databases in realtime systems. A realtime database can be introduced to provide cleaner interfaces and separation between tasks, and a configuration database can be used to support the management of modifications in the system.

A **realtime database** contains global data and constitutes an integral component of any realtime system [Andriopoulos 84, Schoeffler 84, Fisher 83, Sloman 83, Kahn 83]. All the manufacturers of distributed realtime systems have recognised the need for global data, so they treat data management and communications as major issues in the design of their systems [Kompass 84, Pearce 84, Houser 83, Crichton 83a]. According to [Frost 84], "a distributed database available to all systems of a plant through a highly reliable communications network will form the basis of computer architecture during the next twenty years". The role and requirements for such a database are explained in detail in section 2.3.

A **configuration database** contains the descriptions of the components in the system together with their interconnections. It is needed for both the initial development and the subsequent modifications of the system. Such a database already exists in an informal and inflexible form in existing realtime systems but is also encountered in novel software architectures for distributed systems [Kramer 85, Sloman 85] which attempt to cope with change in a more systematic way. The role and requirements for a configuration database are presented in section 2.4.

2.3 The RealTime Database

The role of a realtime database is to separate the various programs (tasks) in a realtime system from the location and representation of the shared data which they access. In this way, the database provides data coupling and becomes the main communication interface between programs as shown in Fig. 2. Unlike commercial databases however, a realtime database is closely related to the programs because of the close interaction between the various programs of a realtime system. More specifically, the role of a realtime database is:

* to provide a global, logically integrated collection of data which may be physically distributed.

* to facilitate the incorporation of man-machine interfaces in the system, especially when a man-machine interface is built in order to provide centralised control over a number of distributed controllers.

* to allow sharing of data among the monitoring, control and management programs by providing access to global data, so it decouples the data from the accessing programs and the various programs from each other.

* to interface to another, possibly higher-level subsystem such as supervisory control, plant management or to a corporate database.

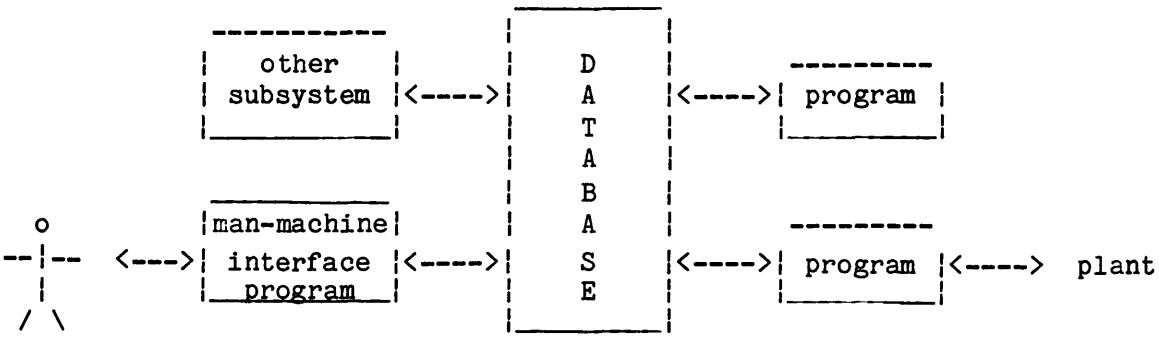


Fig. 2: Simplified View of a RealTime Database

It must be emphasised that the purpose of a realtime database is only to facilitate the design, development, maintenance and overall evolution of the system. The database is not expected to have a direct impact on the human users of the system. Man-machine interface programs will still be used to display the database in the form required by human users. The requirements for such a database will now be explained in detail. Note that most of these requirements are imposed by the realtime environment, are closely inter-related, and in some cases are totally interdependent.

2.3.1 Data Objects

A realtime database contains a variety of data. The time critical data comprises the plant state and the associated display frames. This data typically consists of a few 100's Kbytes, so it is feasible to hold it in main memory rather than on disk. Additional data in a realtime system includes histories, logs and configuration information. This data is not really time critical but is still required to be online and to have fairly fast responses. In this thesis, the term realtime database is

used to encompass all this data rather than be restricted to the time critical data only. More specifically, the data contained in a realtime database may include [Sloman 83, Halling 81, Marsh 79]:

* Plant State: This provides a complete image of the current state of the environment and includes plant variables, parameters, event and alarm indications. Plant state is used primarily for the monitoring and control of the plant.

* Histories & Logs: These include histories of plant variables over past shift, 24 hours, week, month etc, and logs of events and alarms. Histories of production rates, plant delays and failure times are also possible. This information is used for detecting trends, optimising plant performance, plant maintenance and archival purposes.

* Configuration Data: This data is needed for recording the physical layout of the plant as well as the standby and spare components. It can provide online documentation and be used for plant maintenance.

* Display Frames: Such frames are required for displaying information to the human users of the system. A frame contains the parameters and format for the presentation of information as well as the data to be displayed, usually in terms of static background and dynamic foreground data [Marsh 79]. Strings will have to be provided for displaying names, comments, engineering units, alarm messages, etc.

Plant variables can be discrete or analogue. A discrete variable refers to status or digital data which may be a single bit, a group of bits, a status word for a device, binary data, etc. An analogue variable has a numerical value and corresponds to continuous or measurement data. A variety of parameters can be associated with such variables: sensor address, scan period, sensor reading, scaling and linearisation factors, offsets, current value, upper and lower limits, desired value or setpoint, actuator output, etc. Events are system defined incidents of interest or importance, such as the termination of a plant operation. Alarms are special events indicating deviation from normal operation and must be reported immediately to the human users of the system. Examples of alarms are: incorrect status for a discrete variable, the current value of an analogue variable is substantially different from the setpoint or outside limits, failure of some plant equipment, etc.

A realtime database can be visualised as a collection of **data objects.** A data object may be simple or structured. For example, a sensor reading or an upper limit is a **simple** data object, whereas a history or log is a **structured** data object. The type of a simple data object can be integer, real, boolean, character or string. Structured data objects must be able to cover a wide range of data, for example, from a small set of parameters to an entire history. Flexible data structures are therefore required and, in some cases, variant records are necessary. Depending on the size of the plant and most importantly on the amount of history and logs required, the size of a typical realtime database is estimated to be 1-10 Mbytes. One can contrast this with most commercial databases which require 1-10 Gbytes (or even more) of storage.

Data objects may sometimes have to be derived from other data objects. An analogue variable which is not directly measurable may be calculated from other known quantities. A sensor reading can be converted to a more familiar scale of engineering units. Combinatorial events can be defined in terms of simple events, for example, an aggregate alarm. It is thus necessary to provide facilities for **data derivation** based on a mathematical formula or a simple algorithm.

The database should not only allow sharing of data objects at run time but also sharing of **data types** at specification time in order to promote consistency and achieve system-wide representations for standard types of data objects. Unlike commercial databases, a realtime database uses a number (e.g. 40-80) of data types but relatively few instances (e.g. 5-100) of each data type [Crichton 83b]. It should also be possible to supply **initial values** for individual data objects and to specify the geographical location of all the data objects in the system. Default values should be provided for all the options that are left unspecified.

2.3.2 Clients

The term client is used to refer to any program or human user accessing and manipulating data objects from the database. A variety of programs can exist in a realtime system, performing functions such as measurement and actuation, direct digital control, sequence and multivariable control, process identification, energy optimisation, plant management, etc. Man-machine interface programs also exist to allow human users to

access the database and interact with the system.

Several classes of human users can exist such as operators, managers and engineers [Sandoz 82, Halling 81]. They can be classified further as production personnel and support personnel. The production personnel are responsible for the day-to-day running of the plant in order to meet scheduled production targets. The support personnel are responsible for the location and repair of faults (e.g. defective sensors and broken valves), the improvement of the plant performance, and the overall maintenance and evolution of the plant. Human users tend to access the database in a hierarchical fashion, starting with overview (summary) information and proceeding to more detailed information. The database must be structured in such a way that provides a convenient means of generating the desired displays for human users [Williams 78]. No casual human users exist in such an environment, and only a few (e.g. 1-15) human users will be accessing the realtime database at anyone time.

Clients seem to be organised hierarchically: programs are allocated to specific levels in the hierarchy according to the functions they perform, and human users are allocated to specific levels according to their responsibilities and assignments [Williams 78]. They will require different **views** of the database and sometimes may access quite disjoint sets of data objects. Control programs require rather limited views, whereas man-machine interface programs and consequently human users require fairly broad views of the database.

2.3.3 Access & Manipulation

The basic operations are reading and writing data objects. The majority of operations refer to simple data objects such as sensor readings, setpoints, alarms, etc. The same read and write operations apply to structured data objects such as histories and logs, so flexible units for the access and manipulation of data should be provided. There is also some need for compound, application-oriented operations defined in terms of basic operations on individual data objects, for example, an operation that clears an entire log. These operations can take the form of transactions performed atomically on the database and are usually of short duration (not long-lived).

The accesses to the database are known in advance because of the static structure of the system. Access to a data object is by name rather than by content. Transactions are predefined as "canned transactions" [Tripkovic 83], but new transactions can be added at reconfiguration time. Accesses by human users are predetermined and there is no need to have an interpretive database language for queries and updates. Most of the accesses occur at frequent intervals and exhibit a fixed and repetitive pattern because of the regularity of the periodic functions. Bursty accesses should be expected in emergency situations.

2.3.4 Response Times

The response times for accesses to data objects form a continuous spectrum ranging from 100's μsecs or a few msecs for close loop control, to 0.5-1.0 secs for alarm annunciation, to 0.5-5.0 secs for accesses by human users. These responses must be predictable and guaranteed. Only plant state and the associated display frames require such time critical responses. Somewhat slower response times can be tolerated for accesses by human users to histories, logs, configuration data, and their associated display frames. In case that the database is distributed and the required response times cannot be met everywhere, facilities should exist for defining and updating **efficiency copies** of data objects.

2.3.5 Reliability

Similar to response times, the reliability characteristics of the various data objects form a continuous spectrum. In case of failure, plant state should be recovered within a few 10's-100's msecs whereas recovery of histories and logs can take longer. Recovery from failure is usually achieved as a combination of backward and forward recovery [Leveson 84, LeLann 83]. **Backward recovery** restores the data to an earlier consistent state. This is appropriate for histories, logs, configuration data, and the background data of display frames. It is also appropriate for some plant parameters which are specified by human users, such as scan periods, factors, offsets, etc. **Forward recovery** constructs a consistent state of the data from the current state of the plant. This is appropriate for sensor readings and the foreground data of display frames. High reliability can be achieved by replication, so facilities should exist for defining and updating **reliability copies** of data, especially if the database is distributed.

2.3.6 Integrity

Integrity requires that the realtime database is consistent at all times and reflects the state of the environment as accurately as possible. Consistency can be enhanced by ensuring that the data preserves some application-oriented semantics, for example, the fact that the upper limit must not be less than the lower limit, or that a scan period must be positive. Such **integrity constraints** are necessary on the values of the data objects because of the crucial safety requirements for realtime systems. The inherent concurrency in the system requires that concurrent accesses to data do not leave the database in an inconsistent state. Thus, concurrency control mechanisms are needed for synchronising access to the database by many concurrent clients.

The database must remain consistent in the presense of failures. The normal requirement for consistency in case of failure is to ensure that the database contains the latest state of the environment [Crichton 83b, Jervis 84]. This can be achieved by a combination of backward and forward recovery. If the database is distributed, an additional requirement is that the efficiency and reliability copies of data must remain mutually consistent or eventually converge to the same value.

2.3.7 Flexibility

The database should be able to adapt to change and to undergo unlimited incremental expansion. Both modularity and data independence contribute to flexibility. Modularity dictates that the database is designed and built as a collection of standard components with well-defined interfaces between them. Data independence may be defined as the immunity of programs to changes in the location, representation, storage and access strategies of the database. This is achieved by means of levels of indirection and requires that the definition of the data is separate from the definition of the programs that access the data. Since data independence tends to increase response times, provision should be made for optional bypassing mechanisms in order to achieve fast response times. Facilities should also exist for the online generation and the dynamic reconfiguration of a realtime database.

2.3.8 Security

Security restrictions in the form of access rights are required for different classes of clients, especially for human users [Halling 81, Hansen 78]. For example, an operator should not have the same access rights as an engineer. Access rights must be personnel bound and not to the man-machine interfaces that perform the access [Halling 81], so the database should be able to accomodate different access rights for different classes of clients. Note that both integrity and security are protection mechanisms for the database. The difference is that integrity provides protection against accidental corruption of the database by authorised clients, whereas security protects the database against disclosure, modification and corruption by unauthorised clients.

2.3.9 Distribution

A realtime database can be centralised or distributed. A centralised database requires that all the data is transferred to a single physical node where it will be kept. This allows a consistent snapshot of the entire plant state to be easily obtained. It puts however a significant burden on the communication system and suffers from reliability and extensibility problems. A distributed database then becomes a serious alternative. It allows data to remain distributed and only moves data when it is needed elsewhere. Distribution makes it easier to achieve the requirements for response times, reliability and flexibility.

2.3.10 Hierarchical Structure

There is a close relationship between the hierarchical organisation of a realtime system and its database [Halling 81]. The database consists of hierarchical clusters of inter-related data objects exhibiting locality of reference. Such a hierarchical structure has some interesting characteristics. Data is only selectively directed up and is not available everywhere. Raw data is increasingly used at the lower levels, whereas more derived data and historical information is used at the higher levels. The flow of data increases and the response times decrease at the lower levels. Failures are more obvious or even disastrous at the lower levels, so higher reliability is required at the lower levels. Modifications and reconfigurations become more frequent at the lower levels. Hierarchical naming conventions should also be allowed

for data objects reflecting the hierarchical organisation of the plant [Barson 82]. It seems attractive to use the same hierarchical structure for both structuring and naming the data objects of a realtime database.

2.3.11 Support Tools

A number of computer-aided tools are envisaged for the management of a realtime database, such as tools for the online generation, the dynamic reconfiguration, and the interfacing of the database with the rest of the system. A **data dictionary** [ANSI 78, Gray 78] is also required for documenting the data objects and for storing the descriptions and the configuration of a realtime database. Such a data dictionary is similar to a configuration database which is discussed in the next section.

2.4 The Configuration Database

System management constitutes an important aspect of every realtime system. It entails the long-term support activities which are necessary to keep the system operating and allow it to **evolve** [Sloman 81 & 82]. More specifically, system management is concerned with:

i) Configuration: The installation and modification of hardware and software components for system evolution, maintenance or optimisation.
ii) Maintenance: The detection of faults and failures, and the isolation, replacement or repair of faulty and failed components.
iii) Performance Measurement & Optimisation: Measurement of system performance under operating conditions and attempts to improve it by tuning parameters, dimensioning resources, etc.

One of the system management functions will be to maintain an up-to-date image of the entire system. This will require facilities for gathering information from the actual system and for recording it in a management database. Based on this information, system management will make decisions, for example, reconfiguration in the presence of failures. Thus, system management operates on both the system itself and the management database. Since this management database is mainly used for configuration purposes, it is often referred to as configuration database [Sloman 81 & 85, Kim 84]. Its role is to provide the storage and retrieval facilities for the various descriptions and the

configuration of the system, and also to facilitate the exchange of information and interaction between a number of software tools.

2.4.1 Data Objects

The data objects that may be contained in a configuration database are:

* Type Definitions: The database can be used to store the definitions of data types, message types, module types, module interfaces, etc. Definitions may be both in the form of source text and in the form of symbol tables generated by a compiler or other software tools.

* Modules (Programs): This refers to the source text and object code of both system modules (utilities) and application modules (programs). This is traditionally known as "software bank" or "program library".

* Logical Configuration: It describes all the modules in the system together with the interconnections between them. The description of a module includes information such as module name, type and version number, parameter values, system identifier for this module, its connections to other modules, a brief description of its function, etc.

* Physical Configuration: It describes the hardware components in the system (physical nodes, communication media, etc) together with their current status (e.g. operational or failed). The description of a physical node includes information such as the type of its computer(s), memory available, plant interfaces, communication interfaces, physical address in the system, etc. In some systems however, no distinction is made between logical and physical configuration.

* Logical-to-Physical Mapping: This provides the mapping between the logical and physical configurations, for example, by associating the symbolic names of modules with physical nodes or addresses in the system. This is similar to the more general function of locating named objects within a distributed system, for example, see [Oppen 81].

As it has been mentioned already, one of the facilities required for the management of a realtime database is a **data dictionary** [ANSI 78, Gray 78]. This data dictionary will contain most of the configuration data objects mentioned above as meta data about the components of the

realtime database. Hence, a data dictionary and a configuration database may be seen as different manifestations of the same problem and a uniform solution should be provided for both of them.

2.4.2 Other Requirements

A variety of clients will require access to a configuration database, for example: a compiler trying to produce a new symbol table; a system engineer who builds and configures the initial system; a site engineer who performs routine modifications; or a maintenance engineer who examines fault information. The database should be flexible to filter out the relevant information for each kind of client, thus providing views for different kinds of clients and allowing selective retrieval of information. Facilities should exist for grouping together related data objects for naming purposes (e.g. data types), or for configuration purposes (e.g. modules). Most of the data objects are related to each other in various ways and display complex dependencies between them. For this reason, facilities should be provided for accomodating multiple relationships between data objects.

The database may have to keep histories of previous configurations and module versions, so its size can run up to several 10's of Mbytes. Flexibility is also important. The database must allow easy extensions and modifications. It will also have to provide synchronisation between many concurrent clients and to interface with various software tools such as compilers, editors, linkers, loaders, simulators, analysis and validation tools, etc. Thus a configuration database is very similar to a database required for a software engineering environment [Penedo 85, Hooper 85]. Clearly then, a configuration database is not an exclusive need of realtime systems, but represents a more general need related to the management of all computer systems.

Since the configuration database is concerned more with the long-term storage and retrieval of information, response times are not critical but the database is still required to be online for supporting the dynamic reconfiguration of the system. Similar to response times, reliability is not a major issue. A temporary failure can be tolerated, provided that the database can be restored to a recent and fairly accurate state. Consistency is more important than response times and reliability. The configuration database must not contain contradictory

information and must reflect the state of the system as accurately as possible. Security restrictions should be imposed as only authorised people should be allowed to inspect and (re)configure the system.

A configuration database can be centralised or distributed. The obvious and easier to implement solution is to have it centralised. As technology advances however, distributed administration may become possible and consequently the configuration database will have to be distributed [Oppen 81, Sloman 85]. It is likely that hybrid approaches, partly centralised and partly distributed, may appear in the future.

Note that the boundary between a realtime database and a configuration database may appear rather hazy. One could argue then that these two databases should be treated together as a single database. This thesis however suggests that the configuration database should be treated separately from the realtime database because their differences far outweigh their similarities: differences in the kind and size of data objects, in the access patterns, and in the requirements for response times, reliability, distribution and dynamic reconfiguration. The thesis also suggests that a semantic model is the best candidate for such a configuration database. This issue is discussed further in section 3.3.

2.5 Chapter Summary

Realtime systems monitor and control an environment (plant or process) via sensors and actuators. Response times must be fast, predictable and guaranteed. High reliability and availability, ease of extension and change, and dynamic reconfiguration are required. Most designers have structured their realtime systems as a fixed collection of concurrent tasks, exploiting the inherent concurrency and static structure of the environment. Technological advances during the last decade allowed realtime systems to be implemented as distributed configurations which can directly match the physical distribution of the environment.

There is an increasing need for a **realtime database** in most realtime systems. Its role is to separate the accessing programs from the storage and retrieval of the shared data within the system, and to make these programs independent of each other. This separation can bring significant benefits upon the development and maintenance of realtime systems. The database contains the current state of the environment,

histories, logs, some configuration data, and display frames. It must provide facilities for flexible data structures, data derivation and initialisation, and views for many different clients. In addition, it must satisfy the obvious requirements for response times, reliability, availability, extensibility, modifiability, dynamic reconfiguration and distribution, which are normally associated with realtime systems.

A **configuration database** is also needed to help with the management of a realtime system. Its role is to provide the storage and retrieval facilities for the various descriptions and the configuration of the system, so it is very similar to a data dictionary and to a software engineering database. Unlike a realtime database, its response times and reliability are not critical. A configuration database need not be distributed and dynamic reconfiguration is not required. It must provide facilities for data objects with complex relationships between them, support selective retrieval of information, accomodate views for many different clients, be able to interface to a number of software tools, and remain consistent and up-to-date at all times. This thesis treats the configuration database separately from the realtime database because their differences far outweigh their similarities. A semantic model seems to be the best candidate for a configuration database and this suggestion will be supported further in the remainder of the thesis.

CHAPTER 3

RELATED WORK

This chapter provides a survey of related database work. The database models that have been developed for commercial databases are described first: network, hierarchical, relational and semantic. They are followed by the ANSI and NBS architectures that can be used as reference models for discussing and developing database management systems. The database models that have been developed specifically for distributed realtime systems are presented last: the CERN SPS model, the global broadcast model, and the CUTLASS model. The main features of each of these models are carefully explained and then critically examined with regard to their relevance and limitations for realtime systems.

3.1 Network & Hierarchical Models

The network model [ANSI 84a, Taylor 76] provides two data structures: record and set. A record is the basic unit of manipulation of data and consists of a number of fields. A field can be of a simple type (integer, real or string) or a multidimensional array of a simple type. A record is created as an instance of a record type. A set is a collection of records, where one record is designated as the owner and all the others as the members of the set. A set is viewed as an instance of a set type. Each set type has exactly one record type as its owner record type and one or more record types as its member record types.

Integrity constraints can be specified either for a record type or for a set type. A set exists to maintain inter-record relationships. Records are allowed to be members of several sets, so sets are linked together and inter-related. A record can be connected to or disconnected from a set, but can be a member of at most one set of a given set type. The normal way of accessing records is by navigating through sets by following links between records. A network database therefore consists of a number of sets. The network model is data structure oriented, so it is especially suitable for databases which are rather static and have high volume of record-at-a-time processing.

The network model is widespread because it provides flexible structures for modelling commercial databases and its implementations can achieve efficient processing of logically sequential records. However, it is unsuitable for realtime databases for several reasons. It does not provide any facilities for structured data within a field, variant records, boolean datatypes, or derived data. It can treat record types but not records (instances) at specification time, thus different initial values cannot be specified for individual records. Hierachical naming of database components and access to a record by name are difficult to emulate and expensive at run time.

No facilities exist in the network model for mapping records across a number of physical nodes or for defining efficiency and reliability copies of data. In fact, to fragment and distribute a network database is a very difficult task. No dynamic changes are supported, so dynamic reconfiguration is not possible. The performance of a network database is still questionable with regard to realtime processing of records [Crichton 83b]. Note that it is almost impossible to express the CUTLASS model (see below) in terms of the network model [Jervis 83]. The hierarchical model [Tsichritzis 76] can be treated as a special case of the network model, thus for the reasons mentioned above, it is also unsuitable for realtime databases.

3.2 Relational Model

The basic data structure of the relational model [ANSI 84b, Codd 70 & 82] is the table. A relational database consists of a number of tables. A table is an unordered collection of records that may vary over time. A record consists of fields of type integer, real or string. Unlike the network model, the relational model does not support arrays as fields. A field is the smallest unit of manipulation of data. All the records in a table have identical structure. Tables can also be defined as views from other tables. Integrity constraints can be associated either with a single table or with multiple tables. Several operations can be applied orthogonally to any table of the database. The relational model as compared with the network model, depends more heavily on operations than data structures, so it allows many-records-at-a-time processing and provides great flexibility in handling dynamic databases.

The relational model enjoys popularity because of its simplicity, its high level of data independence and flexibility, its strong mathematical foundation, and the multitude of powerful, easy-to-use and friendly end-user languages that accompany it. For these reasons, it is increasingly used for commercial databases. However, it does not support structured data within a field, variant records, boolean datatypes, or data derivation on the fields of a record. It cannot treat and initialise individual records at specification time. Hierarchical naming and access to a record by name are expensive to emulate at run time.

The issues of distribution and efficient realtime processing of records are not addressed and are yet to be resolved by many implementations of the relational model. A relational database however is much easier to fragment and distribute than a network database. Facilities are provided for dynamic changes, for example, creation, deletion and alteration of tables. Such dynamic changes are typical of the fact that the relational model is more biased towards interpretation (the network model is more biased towards compilation). Thus limited dynamic reconfiguration is possible but still has to be demonstrated for distributed relational databases. The basic criticism of the relational model when it is applied to a realtime database, is that it assumes that the cardinality of a table will change frequently at run time, whereas a realtime database is usually of fixed cardinality and changes only at reconfiguration time. For these reasons, the relational model is rather unsuitable for realtime databases.

Nevertheless, the relational model could be used for modelling the configuration database required for system management purposes. An attempt to analyse and model part of such a configuration database in terms of relational concepts can be found in [Mitchell 83].

A number of designers claim that they have developed their realtime databases based on the relational model. The databases of [Koller 81, Juhasz 83, Tripkovic 83] are centralised, and the databases of [Tillman 82, Delatore 85] are distributed. These designers however seem to have used the relational model primarily as a description and design tool, and do not seem to use the operations of the relational model at run time. Their run-time accesses to the database are restricted to individual records and follow predetermined access paths. The advent of fast and reliable database machines may allow relational databases to be

used in realtime systems, for example, Cameo [Pygott 83] is a purpose-built machine designed to support a relational database for military realtime systems. An interesting discussion on extending Modula/R [Reimer 83] by means of an "equate" construct that allows fields of relational records to be accessed as program variables, can be found in [van der Stok 84a & 84b]. Two attempts to use the relational model for modelling both the realtime database and the configuration database can be found in [Tripkovic 83] and in [van der Stok 83 & 84a & 84b].

3.3 Semantic Models

A wide variety of semantic models [Senko 75, Chen 76 & 80, Pirotte 77, Cattell 79 & 83, Hammer 81, Smith 82] have been introduced in order to supplement the semantic inadequacies of the models mentioned above. The basic concepts are those of an entity, attribute and relationship. An entity is a conceptual or real world object about which information must be gathered. An entity may comprise a number of attributes and its interactions with other entities are captured by relationships. These concepts are general and powerful enough to model almost any kind of data. An important advantage for semantic models is the existence of clear and concise graphical representations which can be converted automatically to or be obtained automatically from prose notations. The entity-relationship model [Chen 76] is the most widely used semantic model. Its primary use has been as a database design tool but some prototype implementations have also been developed.

The major criticism for semantic models is the lack of simple and easy-to-use constructs, and the lack of operations with both useful and rigorous semantics. Most of the limitations mentioned for the network and relational models also hold for the majority of the semantic models. An additional disadvantage is that the existing implementations suffer from severe inefficiency problems, especially with regard to response times. For these reasons, semantic models are not suitable for realtime databases. However, as more efficient implementations are developed and as increasing use is made of fast, reliable and intelligent database machines, it may become possible to use such models for man-machine interfaces, for example, operator interfaces [Cattell 79 & 83].

Because of its emphasis on entities and relationships, a semantic model is a suitable candidate and better than the relational model for the configuration database of a realtime system. This suggestion is supported further by the fact that the entity-relationship model is increasingly used for data dictionaries [Marti 83, Chen 80], and also by evidence from many other researchers [Teichroew 80, Cattell 83, Ludewig 85, Penedo 85, Hooper 85]. In practice, the implementation of a semantic model for a configuration database will require an escape exit to a file system for storing source text and object codes efficiently.

It must be emphasised, however, that part of the above criticism towards the network, hierarchical, relational and semantic models is somewhat unfair because these models were never intended to handle issues such as response times, reliability, distribution, and dynamic reconfiguration. Nevertheless, such issues cannot be suppressed and must be addressed explicitly in the context of realtime databases. Probably many of the limitations of the above models stem from the fact that the majority of realtime databases are or will be distributed [Jervis 83]. More details about the limitations of these models can be found in [Andriopoulos 83, Crichton 83b, Jervis 83 & 84, van der Stok 83 & 84a].

3.4 ANSI Architecture

The ANSI architecture [ANSI 78] moves away from the description of data and attempts to provide a reference model for database management systems. The initial objective was to determine which, if any, aspects of such systems were good candidates for standardisation. The main emphasis is on data independence as a way of coping with the inevitable change and continuous evolution of a database. ANSI proposes a three level architecture comprising:

i) Several External Views: An external view is a simplified description of the real world as seen by one or more application programs.
ii) One Conceptual View: The conceptual view is a limited description of the real world maintained for all the application programs in the system.
iii) One Internal View: The internal view is the data representation of this limited description of the real world.

Each of these views are formally defined in the external schemas, conceptual schema, and internal schema respectively. These schemas must be consistent with each other. The placement of the conceptual schema between an external view and the internal view is necessary to provide the level of indirection essential to data independence. In other words, the conceptual schema serves as a common point of agreement and as a stable platform to which external and internal schemas can be bound. The internal schema, for example, may change to reflect new technologies, but the conceptual schema will not be affected. The architecture is independent of any particular database model, so any database model could potentially be used to define a conceptual schema.

The correspondances between the external, conceptual and internal schemas are established through mappings and bindings. Mappings associate descriptors in one schema to those in another. Mappings are also a vehicle for achieving data independence by allowing changes at a given level to be absorbed without affecting other levels. A binding may be seen as the execution of a mapping. The later the bindings are made, the more interpretive and more flexible the system tends to be. A range of bindings is possible, each binding being a different tradeoff between performance and data independence. The architecture also provides mechanisms to bypass levels for performance reasons, thus compromising the data independence, integrity and security of the system.

Furthermore, a data dictionary was identified as an integral part of this architecture. A data dictionary is a meta database, a repository of information about the database. At a minimum, it contains schema and mapping definitions. It may also include usage statistics, security restrictions, descriptions of users, etc. A data dictionary is therefore similar to the configuration database of a realtime system.

The basic criticism for the ANSI architecture is that it is not concrete enough or precise. At best, it is a reference model for understanding, describing and discussing database management systems. It does not address any of the issues of response times, reliability and dynamic reconfiguration. It seems to apply primarily to centralised databases, but several attempts have been made to extend it for distributed databases, for example, [Toth 78, Adiba 78, Litwin 79].

3.5 NBS Architecture

The NBS architecture [NBS 82, Manola 83] aims at unifying the wide variety of functions performed by commercial database management systems. Its intended use is as a reference model for understanding and developing database management systems, for accomodating multiple database models, and for defining database standards. The architecture identifies a number of components, the functions performed by each component, its external interfaces to human users or application programs, and its internal interfaces to other components. Each component therefore represents a group of related functions and is characterised by a set of interfaces via which its functions can be invoked. Four major components have been identified:

i) Schema Processors: A collection of facilities for the definition of data, views, integrity constraints, security restrictions etc.
ii) User Interfaces: Access interfaces for query languages, host languages, report generators etc.
iii) Database Administration Aids: The aids for the management of the database, i.e. data dictionary and database design tools.
iv) Core Database Handler: The central part of a database management system that is concerned with the storage and retrieval of all the data in a system, i.e. both normal data and meta data.

For each of these major components, constituent components have also been identified. The NBS architecture may be seen as continuation and partial elaboration of the ANSI architecture, but the emphasis is on functional components rather than on data independence and separate levels of description. It goes into greater length and detail in some areas, so it appears to be at a more concrete level and more complete in these areas. Similarly to the ANSI architecture, it does not address the issues of response times, reliability and dynamic reconfiguration. Moreover, it seems to be appropriate only for centralised databases. It is hard to envisage how it can be extended for distributed databases.

3.6 CERN SPS Model

This model is used for the monitoring and control of the CERN 400 GeV Super Proton Synchrotron (SPS) [Crowley-Milling 74 & 79, Altaber 81, van der Stok 83]. The concept of a data module is at the heart of the CERN

SPS model. A data module is an interface routine, i.e. system software that provides data derivation and control abstraction for a specific device. In this way, it presents a high-level interface to application programmers, simplifying the access to hardware devices by application programs. A data module is associated with each kind of device. Each data module consists of a device driver and a data table containing the parameters associated with the corresponding device. Possible parameters are: current value, setpoint, output value, conversion factor, linearisation factor, upper limit, lower limit, hardware address, protection keys, etc. Monitoring and control of the synchrotron is achieved by reading and writing (setting) these parameters. Note that for some parameters reading or writing may also imply hardware access, whereas for others it may only refer to data held within the data module. A data module usually resides in main memory. For a variety of reasons including sharing of code for the device driver, a data module may handle a number of similar devices. Since most of the devices are geographically distributed, the collection of all the data modules forms the CERN SPS distributed database.

Access to a parameter from an application program is by stating the generic name and serial number of the device, the parameter required, and a new value for write operations. The reply contains a return code and a value for read operations. All bindings are carried out at run time. In addition, the data tables of all the data modules are periodically copied to a centralised database in order to provide a rapid recovery backup in case of failures, and to act as a source of information for surveillance, logging and analysis programs. The CERN SPS model, although it constituted an important breakthrough at the time it was developed, has certain drawbacks: it fails to separate the control from the database functions, its view of a realtime database is very limited, it is rather device or hardware oriented, and dynamic reconfiguration is not always possible.

3.7 Global Broadcast Model

Many distributed realtime systems incorporate a limited database model based on global broadcast data [Crichton 83a, Gueth 83 & 85a & 85b, Houser 83, Kompass 84]. This model assumes that the physical nodes of a distributed system are interconnected by a broadcast communication system. The "owner" of a data object broadcasts its current value to

all physical nodes, periodically or on change. A "user" of a data object registers its interest in a particular object and so receives the most up-to-date value for that object into a local copy. Updates for a data object are always sent to and are carried out by the "owner" of the data object. This model is rather implementation oriented and very low level in that it deals with simple data objects. It is not extensible to the generalised data structures required for histories and logs. It relies entirely on the notion of a fast broadcast communication system. In existing implementations this is usually a single serial highway. A large distributed realtime system will require multiple interconnected subnets [Kramer 83, Sloman 84]. It is impractical to consider broadcasting of the complete database for a large system.

3.8 CUTLASS Model

The CUTLASS model [Barson 82, Bransby 82, Jervis 83 & 82] defines a distributed database that is used for the monitoring and control of electricity generation plants. This database contains all the data necessary for the control of individual parts of an electricity generation plant and is the source of all information presented on supervisory displays. All the application programs are essentially structured around this database. In order to achieve the response times required, the database resides in main memory and is tightly coupled to the application programs and to the rest of the CUTLASS system.

A data object can be either of a simple type (integer, real, boolean or string) or an one-or-two dimensional array of a simple type. A data object can also be derived from other data objects. Levels of access and security can be specified for each data object. Related data objects are then grouped together to form a record. Such a record is used as a record type out of which several records are created. Initial values for individual records can be specified. A number of records of the same record type can be grouped together to form a record class. The logical structure of the database is a collection of record classes. Each record class is subsequently decomposed into partition classes, a partition being the unit of distribution of data. Efficiency and reliability copies for the data objects within a partition class can be defined at this stage. Each partition class is then allocated to a physical node within the network. The physical structure of the database is a collection of partition classes. Access to a data object is by name. The

binding is normally done at compile time but it can be deferred and be done at run time instead. Facilities exist for defining a hierarchical name structure and for assigning names to individual records.

The CUTLASS model is a nice generalisation of the global broadcast model. The resulting model is more abstract, attempts to separate the logical data structures from the physical distribution of the database, provides more flexibility and data independence, and most importantly is formalised in terms of a schema definition language. It is certainly an important breakthrough in the evolution of realtime databases and is indicative of the current state-of-the-art. However, its data definition facilities are limited (no arrays of records, no variant records, cannot easily model a history or log) and its data access operations could be more general. The separation between the structural definition of a database and the assignment of names to database components is confusing and leads to unnecessary proliferation of names. The definition of a schema is rather monolithic, modifications are limited, and dynamic reconfiguration is not always possible.

3.9 Chapter Summary

This chapter presented the main features of existing database and proposed reference models, and critically examined these features with regard to their relevance and limitations for realtime systems. The network, hierarchical, relational and semantic models are not suitable for a realtime database: their structures are inappropriate for such a database and they fail to address the important requirements for response times, reliability, distribution and dynamic reconfiguration. However, the relational model is suitable and a semantic model is recommended for the configuration database of a realtime system. The ANSI and NBS architectures are neither very relevant nor of much use. The CERN SPS model and the global broadcast model are relevant but they are low level and implementation oriented and their view of a realtime database is very limited. The CUTLASS model is certainly an important breakthrough in the evolution of realtime databases, but it could be more general with regard to data structures, modifications and dynamic reconfiguration. For these reasons, a new and improved realtime database model is presented in the remainder of this thesis.

CHAPTER 4

REALTIME DATABASE MODEL

This chapter proposes a new database model especially suited for realtime databases. The basic concepts of the model are overviewed first. Then a notation for specifying individual database components is described. This is followed by a notation for specifying a database out of individual database components. An extension to the model for performing database modifications is also presented. The proposed concepts and notation are illustrated by means of examples referring to the realtime database of an electricity generation plant.

4.1 Model Overview

The model is a synthesis of ideas and experiences from realtime systems, distributed systems and databases, application-oriented specifications, programming languages and software engineering. It provides the concepts and a notation for the specification of a realtime database that is expected to be readily implementable, i.e. with minimal transformation.

The model proposes a **data module** as a distributable unit of shared data in a realtime system. A data module may represent plant state, a history, an operator display frame, or simple configuration data. Data modules complement the **processing modules** (tasks) which perform the various processing functions in a realtime system. Typical processing modules read data from data modules or devices, perform some processing, and write data to data modules. Processing modules may also communicate directly by message passing or remote procedure calls [Andrews 83, Birrell 84, Nelson 81]. The specification of processing modules is outside the scope of the database model and a suitable language such as Conic [Kramer 85, Sloman 85] may be used for their specification.

A data module specifies **data objects** (such as sensor readings, alarms, histories, logs, etc), operations on the objects, integrity constraints, response times, reliability and consistency characteristics. The basic **actions** on the objects of a data module are reading or writing values. Data modules can hold copies of objects that reside in other modules. These copies are automatically updated periodically or when the value of

the master object changes. A data module can also specify **transactions** which are a collection of actions on local or remote objects performed as an atomic operation. The data module forms the unit of distribution and reconfiguration, the unit of consistency (with regard to integrity constraints and transactions) and recovery for reliability purposes.

The proposed model provides the concepts and a notation for the logical specification of the data modules and their interactions in a realtime database. It attempts to address the realtime database requirements of section 2.3, but does not have the limitations of the database models of chapter 3. It is meant for use during the conceptual and logical design phase of a database, after completion of the requirements specification [Yao 85 & 82, Agosti 84, Teorey 82, Lum 79] (see also section 7.5). The long term aim would be to provide automatic transformation of the modules into implementation components. The current implementation is manual in terms of the components discussed in chapter 6.

A **Data Definition Language** (DDL) is used to specify individual data modules. The DDL allows the specification of different **views** on the data objects in a data module. Views provide interface control on the data and operations encapsulated by a module, thus facilitating the interconnection of the various modules to form a database. A view is a named collection of data objects and operations **exported** by a data module. This allows different, overlapping views (groupings) of the data and operations encapsulated by a data module. A view can be **imported** by another module which then has access only to the data objects and operations specified by the view. A data module may not necessarily store data. It could be merely a grouping of actions or transactions on data held by other data modules.

A **Database Configuration Language** (DCL) is used to specify a **database module.** This contains interconnected instances of data modules and other database modules and so allows nesting of component specifications. The interconnection binds an imported view of a module instance to the exported view of another instance. An extension to the DCL can be used to specify configuration changes. The separation between the DDL (data-modelling-in-the-small) and the DCL (data-modelling-in-the-large) facilitates the development of a realtime database and provides the flexibility in adapting to changes [DeRemer 76, Mitchell 79, Magee 84, Kramer 85]. The database can be decomposed to a set of data modules with

well-defined interfaces between them. Individual data modules are easier to understand, specify, implement and modify. They can also be compiled separately. Hence, the entire database is easier to design, specify, implement and maintain. The DCL records the interactions between data modules explicitly so that modifications can be performed in an orderly and safe manner. The approach is novel in that it can allow dynamic reconfiguration of the database without shutting down the whole system.

4.2 The Data Definition Language (DDL)

The data definition language is used to specify an individual data module type from which data module instances can be created using the database configuration language. All the **module types** are assumed to exist in a global, system-wide pool of module types. Similarly, **data types** which are common to multiple data modules can be imported from a pool of types. These two pools of types may be held in the configuration database of the system.

Fig. 4.1 shows the definition of three simple data types (two integers and one real). One of the integers is constrainted to be in the range 61440 to 65536, and an initial value of 0.0 is supplied for the real. Fig. 4.2 shows the specification of the 'BinaryOut' data module containing four simple data objects (one integer and three booleans). 'BinaryOut' may be used for the control of a digital output, for example, an on/off switch or valve. The 'ControlIn' can be set by an intelligent controller, engineer or operator. An actuator situated at the physical address 'Location' is assumed to output the value of 'ControlIn' when 'Enable' is true. In other words, 'Output' indicates the desired status for the actuator at address 'Location'. This use of the 'BinaryOut' data module is reflected by its three exported views: 'HumanView' used by an engineer or operator to access and manipulate the data objects in the data module; 'ControllerView' used by an intelligent controller to set the desired output; and 'ActuatorView' used by actuator software to output the desired status. These and other concepts that can be specified within a data module, are presented in detail in sections 4.2.1 to 4.2.7. Note that { } is used enclose comments.

```
InstrumentAddress : Integer, values 61440..65536;

Measurement       : Real, initially 0.0;

ScanPeriod        : Integer;
```

Fig. 4.1: Simple Data Types

```
DATA MODULE BinaryOut;

Export  HumanView: View
                     Output:Read;
                     Location,Enable,ControlIn:Read,Write;
                   End;

        ControllerView: View
                          ControlIn:Read,Write;
                        End;

        ActuatorView: View
                        Location,Output:Read;
                      End;

Type    InstrumentAddress;

Data    Location  : InstrumentAddress;
        Enable    : Boolean, initially true;
        ControlIn : Boolean, initially false;
        Output    : Boolean= Enable and ControlIn;

END. {BinaryOut}
```

Fig. 4.2: Data Module With Four Simple Data Objects

4.2.1 Data & Datatypes

A data module can encapsulate a number of related data objects such as sensor reading, scale factor, upper and lower limits, etc, pertaining to a particular sensor. The data objects within a data module can be either local data or secondary copies of data objects from other data modules. A data object is specified in terms of the following:

i) **Object Name** which is used to access the object and must be unique within the scope of the data module.

ii) **Object Type** which can be primitive (integer, real, boolean, byte, character, string or enumerated [Jensen 78]) or structured (array, record, variant record).

iii) **Value Constraint** which specifies the values that can be taken by the data object. This is a value-set or (sub)range constraint.

iv) **Initial Value** of the object when it is created.

The type, value constraint, and initial value of a data object must obviously be consistent with each other. Only the name and type are mandatory. The value constraint and initial value are optional. Suitable defaults will be provided if necessary. This is illustrated by the specification of 'Location', 'Enable' and 'ControlIn' in Fig. 4.2. **Derived Data Objects** are allowed and are calculated from other local data objects, for example, an average sensor value or an unmeasurable parameter which can only be derived from other measurable parameters. They are specified in terms of an expression involving constants and other data objects of the data module. 'Output' in Fig. 4.2 is a derived data object.

Structured objects can be obtained by applying the Array, Record or Case constructors (possibly recursively) to primitive objects. The **Array** constructor facilitates the grouping and naming of identical objects. Indices for arrays can be of any discrete data type, i.e. integer, boolean, character or enumerated. The **Record** constructor is used for grouping and naming of objects that may not be of the same type. The **Case** constructor provides structural variability within an object and can model some forms of existence dependencies. Fig. 4.3 shows two examples of record constructors corresponding to the parameters of a 'DigitalPoint' and 'AnalogPoint' respectively.

The 'Status' data object in Fig. 4.4 demonstrates the combined use of array and record constructors as well as the separate initialisation of each array element. The 'BinaryIn' data module of Fig. 4.4 refers to a collection (probably on the same physical card) of four digital input points. Compared with the 'BinaryOut' data module of Fig. 4.2, the 'BinaryIn' data module also includes response time, reliability and consistency characteristics which are explained in detail in section 4.2.7. The simple, non-derived data objects and the structured data objects will be referred to as **base data objects.**

```
DigitalPoint: Record
                Address : InstrumentAddress;
                Value   : Boolean;
              End;

AnaloguePoint: Record
                 Address    : InstrumentAddress;
                 Value,
                 HighLimit,
                 LowLimit   : Measurement;
                 RangeAlarm : Boolean;
               End;
```

Fig. 4.3: Structured Data Types

```
DATA MODULE  BinaryIn;

Export       HumanView     : View Status:Read       End;
             ControllerView: View Status:Read,Write End;

Type         DigitalPoint;

Data         Status : Array [1..4] of DigitalPoint,
             initially <<64004,true>, <64006,false>,
                        <64008,true>, <64010,true>>;

Performance  Response Time : 10 msec;
             Reliability   : hot backup;
             Consistency   : medium;

END. {BinaryIn}
```

Fig. 4.4: Data Module With One Structured Data Object

A data object can also be declared as an instance of a predefined type, for example, 'Location' of type 'InstrumentAddress' in Fig. 4.2. Types which are common between multiple data and processing modules should be defined in a separate definitions pool and imported into the data module. This allows sharing of data types at specification time and promotes consistency between different data objects of the same type (cf. domain concept in some versions of the relational model).

A value constraint and an initial value can also be supplied for a data type, for example, 'InstrumentAddress' and 'Measurement' in Fig. 4.1. Whenever a data type is used (either for the definition of another data type or for the declaration of a data object in a data module), its value constraint and initial value can be overwritten provided that the

new value constraint is a subset of the old one, and that the (new) initial value satisfies the (new) value constraint. The syntax for data types is identical to the one used for data objects.

Copy Data Objects are local copies, provided for performance (response time) reasons, of primitive or structured data objects residing in other data modules. For example, a particular sensor reading may be accessible as a named data object and also as part of a summary data object. A copy data object does not provide redundancy for reliability purposes, as the unit of redundancy is a complete module. A "single primary update strategy" [Lindsay 79] is supported, so all read actions are addressed to the local copy whereas write actions must be sent directly to the remote primary copy, i.e. secondary copies are read only. The remote primary copy updates all its secondary copies either periodically (**periodic**) or whenever its current value changes (**onchange**).

This single primary update strategy has been chosen because it is conceptually simple and gives good performance for realtime systems [Lindsay 79, Borr 81, Barson 82, Crichton 83a, Houser 83, Cheng 84]. The data module containing the primary data object must specify that it is exported either periodically or on change. The data module containing a secondary copy must import the data object. The declaration of the secondary copy indicates the primary's module type, data object, and update mechanism which must be the same as that specified in the exporting module. An **update period** or **onchange condition** can also be specified at the exporting module, but not at the modules where secondary copies are held. Fig. 4.5 shows the 'CachedValue' data object held in a 'Local' data module, that is updated periodically every second from the 'Value' data object of a 'Remote' data module.

```
DATA MODULE  Remote; {primary}

Export Value:Periodic 1 sec;
. . .
Data   Value:Real; {data object declaration}
. . .
END. {Remote}


DATA MODULE  Local; {secondary}
. . .
Data CachedValue:Real <-- Remote.Value periodic;
. . .
END. {Local}
```

Fig. 4.5: A Copy Data Object

```
DATA MODULE Voltage;

Export EngineerView: View
                       Value:Read;
                       Location,Input,Factor,Offset,Interval,
                       UpperLimit,LowerLimit,RangeAlarm:Read,Write;
                     End;

       OperatorView: View
                       Parameters:Read;
                       UpperLimit,LowerLimit,RangeAlarm:Write;
                     End;

       . . . . .

Type   InstrumentAddress; Measurement; ScanPeriod; AnaloguePoint;

Data   Location  : InstrumentAddress;
       Input     : Measurement;
       Factor    : Integer, values 1..100, initially 100;
       Offset    : Measurement;
       Value     : Measurement = (Factor*Input)/100 + Offset;
       Interval  : ScanPeriod;
       UpperLimit,
       LowerLimit: Measurement;
       RangeAlarm: Boolean, initially false;

       Parameters: AnaloguePoint
                   <Location,Value,UpperLimit,LowerLimit,RangeAlarm>;

Constraints  UpperLimit >= LowerLimit;

Performance  Response Time : 50 msec;
             Reliability   : warm backup;
             Consistency   : medium;

END. {Voltage}
```

Fig. 4.6: Data Module With A Constructed Data Object

Constructed Data Objects specify partially-overlapping data objects and provide different units of manipulation of data within a single data module. For example, a particular data object may be accessible from a data module both as an individual primitive data object and as part of a record. These are conceptually similar to the "type constructors" of various programming languages. If a data object is read only then any constructed data object of which it is a component must also be read only. Fig. 4.6 specifies a 'Voltage' data module with ten data objects. 'Voltage' may refer to the parameters associated with an analogue input point that measures voltage in an electricity generation plant. The specification allows each of the parameters 'Location', 'Value',

'UpperLimit', 'LowerLimit', and 'RangeAlarm' to be accessed individually or all together as the constructed data object 'Parameters' (of type 'AnalogPoint' as defined in Fig. 4.3). Note that 'Parameters' is read only because it contains the derived data object 'Value'. Constant values can also be specified as part of a constructed data object. This facilitates the modelling of display frames. A constructed data object that contains at least one constant value will be read only.

4.2.2 Actions

The data module supports four basic, indivisible operations which can be applied to any data object in the module. It acts as a server for **Read** and **Write** actions and it instigates **Periodic** and **OnChange** actions to update remote copies of data objects. These actions are generic because they can apply to all data objects irrespective of their data types. Write actions cannot be used on derived or copy data objects. The notation for the invocation of Read and Write actions by a client is:

```
Read  (in Data-Object-Name; out Data-Object-Value);
Write (in Data-Object-Name; New-Data-Object-Value);
```

4.2.3 Exports

Different **views** of the data and operations encapsulated by a data module can be specified to give precise **interface control** [Clarke 83]. For example, one view of a data object may be read only whereas another view may allow a write action to it. A view is a named collection of data objects and specified actions on those objects. These are made accessible to a module which imports the named view. A view can also comprise transactions (see later) and views imported from other modules. In this way, "nested views" can be obtained. The different views on a data module may overlap or be disjoint. For example, the 'ControllerView' and 'ActuatorView' in Fig. 4.2 are disjoint. The 'EngineerView' and 'OperatorView' in Fig. 4.6 have some ovelap but the former also allows write actions on 'Location', 'Input', 'Factor', 'Offset' and 'Interval'.

A separate **update period** can be specified for each Periodic action included in a view. Similarly, a separate **onchange condition** can be given for each Onchange action in a view. An onchange condition is a first-order predicate (assertion) involving constants and data objects

specified within the data module. "Old" and "New" values of data objects may be used. An update period or onchange condition is optional. If it is omitted, an appropriate default will be assumed, for example, 1 sec as default update period and any change from the old value as default onchange condition. Separate update periods and onchange conditions within views imply that different clients receive updates for a data object at different times or when different conditions become true.

A view specified in this way combines together "view definition" and "view mapping", according to conventional database terminology. A data object or transaction must be explicitly exported in a view to be accessible. A shorthand notation for specifying views consisting of a single data object or transaction is provided in order to avoid proliferation of names, for example, the 'ControllerView' in Fig. 4.2 could have been defined simply as 'ControlIn:Read,Write'.

4.2.4 Imports

Views are imported from other data module **types**. Imports specify the actions and transactions invoked on other data modules, and updates needed by remote data modules on local copies of data objects. Access to another module is via one of its exported views. An import is specified by associating a local named placeholder with the view exported by another module. An imported view is bound (linked) to an exported view of a data module **instance** using the DCL (see section 4.3). Arrays of imported views with integer indices are also allowed, for example, the imported view 'Valve' in Fig. 4.7 allows access to the data of five different valves.

It must be emphasised that a data module does not necessarily specify data or transactions. It may merely define a view on data objects and operations exported from other modules. Fig. 4.8 shows a data module called 'Views' that defines 'TwoValveSets' as a new composite view in terms of the two imported views 'ValveSet1' and 'ValveSet2'. This 'TwoValveSets' view may be used to unify the manipulation of two sets of valves rather than accessing each set of valves separately.

```
DATA MODULE  ValveManipulation;

Export  TurnOnOff: Transaction;

Import  Valve: Array [1..5] of BinaryOut.HumanView;
       {provides access to 5 instances of a module type}

Transaction  TurnOnOff (in Open:Boolean);
             Var  I:Integer;
             Begin
               For I:=1 to 5 do
                 Valve[I].Write(ControlIn,Open);
             End;

END. {ValveManipulation}
```

Fig. 4.7: Array of Imports & Transaction

```
DATA MODULE Views;

Export  TwoValveSets: View
                        ValveSet1,
                        ValveSet2:View;  {nested views}
                      End;

Import  ValveSet1,
        ValveSet2: ValveManipulation.TurnOnOff;
       {provides access to two instances of a module type}

END. {Views}
```

Fig. 4.8: Data Module for View Definition

4.2.5 Constraints

An integrity constraint specifies a condition to be satisfied by the current state of data objects within the data module. It is specified in a declarative way, i.e. as a first-order predicate (assertion). The predicate must hold before and after an operation is invoked on the data module. The checking and enforcement of a constraint can be triggered by a write operation on a data object involved in the constraint. Integrity constraints cannot be specified for copy data objects. Appropriate choice of integrity constraints can add more application-oriented semantics to the data and consequently increase the integrity of the data objects stored in the data module. An operation which would violate a constraint is rejected (not performed). The constraint in the 'Voltage' data module of Fig. 4.6 specifies that the 'UpperLimit' must

always be greater than or equal to the 'LowerLimit'.

4.2.6 Transactions

Transactions allow consistent snapshots to be obtained and atomic updates to be performed on multiple data objects that may be stored across a number of data modules. A transaction is assumed to have the all-or-nothing property, i.e. either it is executed to its entirety or not at all. It can be used to provide a unit for procedural abstraction [Schmidt 83], semantic integrity, concurrency control and recovery from failures. In this way, transactions can allow application-oriented operations that access and manipulate data to be specified as part of the data module (cf. modelling behavioural semantics of databases [Brodie 81, Ridjanovic 83]). A data module that only exports transactions is essentially an abstract data type [Liskov 74 & 77, Claybrook 85]. Fig. 4.7 shows how the transaction 'TurnOnOff' opens or closes five valves all together as an atomic operation.

The body of a transaction is specified as a Pascal block [Jensen 78], although any other equivalent notation could be used. Parameters are passed by value in and out of the transaction. Local data for workspace can be declared if necessary, using the standard data definitions provided by Pascal, for example, the variable 'I' in Fig. 4.7. Data objects specified in the data section can be used within "with", "case", or "assignment" statements as if they were global data. This means that the assignment statement is overloaded to model read and write actions to and from data objects contained in the data module. A transaction may invoke other transactions, maybe on different data modules. Thus, "nested transactions" can be obtained, cf. [Moss 81, Weber 83]. Remote transactions and actions must be imported via a view, for example, see in Fig. 4.7 how remote write actions are imported and used. A remote invocation is specified in the form:

local-view-name . operation-name (actual-parameter-list)

The statements defining a transaction are assumed to execute serially because of the Pascal semantics, but concurrent execution of statements could also be accomodated. Parallelism is assumed to exist among different transactions. A transaction must satisfy the integrity constraints on all the data objects it updates, or else it will be

aborted. Transactions within a data module are obviously defined statically at specification time ("canned transactions" [Tripkovic 83]). New transactions can be added by installing a new data module, which can be performed dynamically but is assumed to occur infrequently.

4.2.7 Performance

The performance section specifies the characteristics or requirements to be met by an implementation of the data module. This section can be viewed in two ways: either as design guidelines for an effective implementation, or as the acceptance test to validate an implementation of the data module. A performance characteristic can be specified either as a single value or as a range of values. The latter corresponds to tolerances in the expected performance. The performance characteristics that can be specified are: response time, reliability and consistency.

Response Time gives the maximum delay with which a read or write action, or transaction is serviced by the data module.

Reliability defines the kind of backup copy that must be maintained for recovery from failure purposes. Note that reliability copies (backups) apply to a complete data module and are treated differently from efficiency copies (copy data objects). Four levels of reliability can be specified:

i) Hot backup: Failure of a data module is transparent to its clients and the backup copy of the data is always guaranteed to be up-to-date.
ii) Warm backup: A relatively fresh or recent copy of the data is maintained and can be restored after failure, but some updates may be lost.
iii) Cold backup: All data is lost in case of failure and a newly initialised copy is used to accomodate future updates.
iv) None: No backups are maintained.

Consistency specifies the parallelism between concurrent operations in terms of levels of consistency. This effectively allows different tradeoffs between response times and semantic integrity [Gray 76, Eswaran 76, Traiger 83]. Three levels of consistency are possible:

i) Strong: A write operation excludes all other write or read operations on a single data object. Strong consistency guarantees absolute consistency for the data of the data module.

ii) Medium: Several reads and a single write operation on the same data object can proceed concurrently. Medium consistency guarantees no lost updates but allows dirty reads.

iii) Weak: Several read and write operations can proceed in parallel. Weak consistency allows both lost updates and dirty reads.

The data modules 'BinaryIn' of Fig. 4.4 and 'Voltage' of Fig. 4.6 include typical specifications of performance characteristics. The performance characteristics of a data module can be checked for consistency with those of imported modules. This can be done by tracing the various dependencies and checking to see if they are ultimately satisfied or not. Deadline calculations, for example, may be used for checking the response times required [Wirth 77, Kramer 79, Haase 81].

4.3 The Database Configuration Language (DCL)

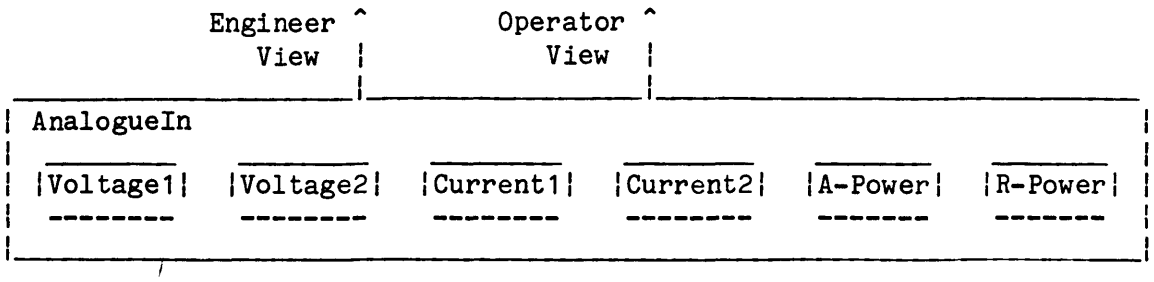
The database configuration language (DCL) is used for specifying a database as a collection of data module instances together with the interconnection of these instances via views. A **database module** is a type from which instances can be declared within another database module. Thus database specifications can be nested.

Fig. 4.9 shows the specification of the 'AnaloqueIn' database module which holds analogue status information for an electricity generation plant. It comprises six input points: two of type 'Voltage', two of type 'Current', one of type 'ActivePower', and one of type 'ReactivePower'. 'Voltage' is the data module of Fig. 4.6. 'Current', 'ActivePower', and 'ReactivePower' are similar to 'Voltage'. 'AnalogueIn' also specifies two views that are used by an engineer and operator respectively to access and manipulate each of these six measuring points. The graphical representation for 'AnalogueIn' is given in Fig. 4.10(a). It can be seen that the concepts and notation for a database module are similar to those for a data module.

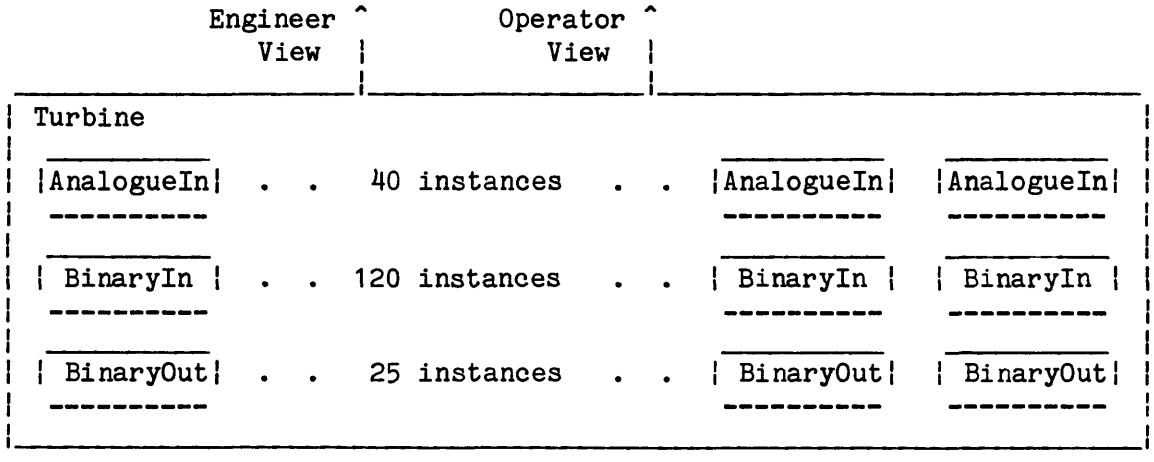
```
DATABASE MODULE AnalogueIn;

Export     EngineerView: View
                           Voltage1.EngineerView,
                           Voltage2.EngineerView,
                           Current1.EngineerView,
                           Current2.EngineerView,
                           ActivePower.EngineerView,
                           ReactivePower.EngineerView:View;
                         End;

           OperatorView: View
                           Voltage1.OperatorView,
                           Voltage2.OperatorView,
                           Current1.OperatorView,
                           Current2.OperatorView,
                           ActivePower.OperatorView,
                           ReactivePower.OperatorView:View;
                         End;

Type       Voltage;
           Current;
           ActivePower;
           ReactivePower;

Instance   Voltage1,Voltage2: Voltage;
           Current1,Current2: Current;
           ActivePower;
           ReactivePower;

Constraint   COUNT(Voltage)=COUNT(Current)

Performance  Response Time : 100 msec;
             Reliability   : warm backup;
             Consistency   : medium;

END. {AnalogueIn}
```

Fig. 4.9: Database Module for Six Analogue Inputs

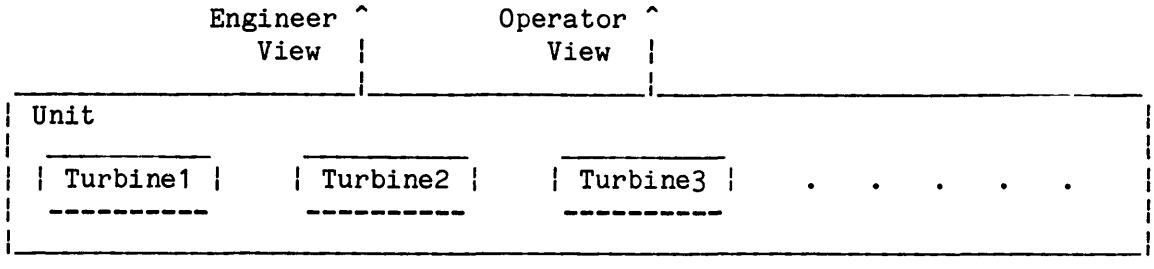
The **type** section of a database module lists the names of module types that are imported from the pool of module types. A module type may correspond to a data module or to a database module. These modules types define a context for declaring module instances. Any number of instances can be created from a single module type. A module instance is created by specifying a name followed by the name of its module type. Whenever only a single instance is created from a given module type, the name of the module type can be reused to denote that instance (thus, avoiding proliferation of names). For example in Fig. 4.9, the instance 'ActivePower' has the same name as the type, but two instances of 'Voltage' are needed so they are named 'Voltage1' and 'Voltage2'.



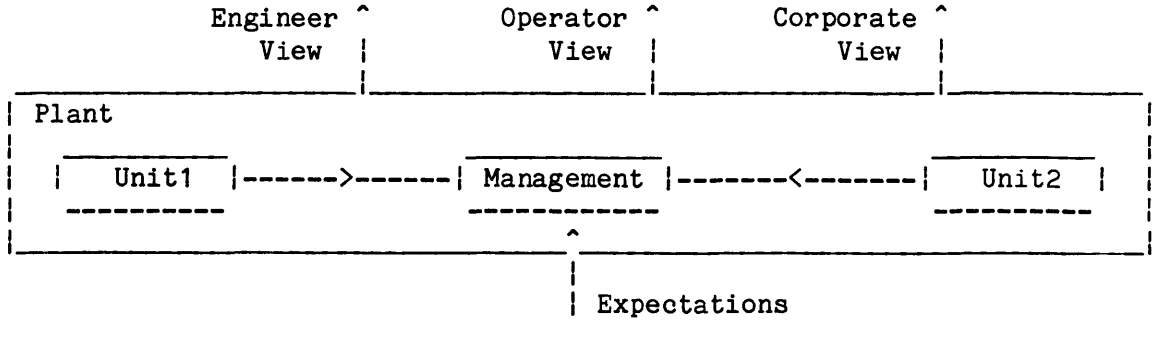
(a)



(b)



(c)



(d)

Fig 4.10: Database Configuration for an Electricity Generation Plant

```
DATABASE MODULE Turbine;

Export     EngineerView: View
                           AnalogueInput[1..40].EngineerView,
                           BinaryInput[1..120].HumanView,
                           BinaryOutput[1..25].HumanView:View;
                         End;

           OperatorView: View
                           AnalogueInput[1..40].OperatorView,
                           BinaryInput[1..120].HumanView,
                           BinaryOutput[1..25].HumanView:View;
                         End;

Type       AnalogueIn;
           BinaryIn;
           ControlOut;

Instance   AnalogueInput: Array  [1..40] of AnalogueIn;
           BinaryInput  : Array [1..120] of BinaryIn;
           BinaryOutput : Array  [1..25] of BinaryOut;

Performance  Response Time : 500 msec;
             Reliability   : warm backup;
             Consistency   : medium;

END. {Turbine}
```

Fig. 4.11: A Database Module With Arrays of Instances

The **array** constructor can be used to create multiple instances from the same module type, resulting in very concise specifications. The 'Turbine' database module in Fig. 4.11 contains 'AnalogueInput', 'BinaryInput' and 'BinaryOutput' which are specified as arrays of module instances. Such arrays can be of any dimension, but for simplicity only integer indices are allowed. Fig. 4.11 also demonstrates how views from arrays of module instances can be exported as nested views. The graphical representation for 'Turbine' is given in Fig. 4.10(b).

```
DATABASE MODULE Unit;

Export     EngineerView: View
                           Turbine1.EngineerView,
                           Turbine2.EngineerView,
                           Turbine3.EngineerView, . . . :View;
                         End;

           OperatorView: View
                           Turbine1.OperatorView,
                           Turbine2.OperatorView,
                           Turbine3.OperatorView, . . . :View;
                         End;

Type       Turbine; . . .

Instance   Turbine1, Turbine2, Turbine3: Turbine; . . .

Performance  Response Time : 1 sec;
             Reliability   : warm backup;

END. {Unit}
```

Fig. 4.12: Nested Views

```
DATABASE MODULE  Management;

Export  Report: View  . . .  End;

Import  Objectives: Production.Targets;
        UnitView1,
        UnitView2 : Unit.OperatorView;
. . . .

END. {Management}
```

Fig. 4.13: Exported & Imported Views for Management

Module instances are interconnected by **linking** (binding) imported views to exported views. Fig. 4.12, 4.13 & 4.14 show the 'Unit', 'Management' and 'Plant' specifications for an electricity generation plant. Fig. 4.14 demonstrates the use of links: 'Management' has two imported views 'UnitView1' and 'UnitView2' which are linked to the exported view 'OperatorView' of each of the modules 'Unit1' and 'Unit2'. The type of an imported view (i.e. module type and view name) must be the same as the type of the exported view, i.e. only type-compatible views can be bound together. Arrays of module instances can also be linked together by specifying either an index specifier (which denotes a single instance) or a range specifier (which denotes a group of instances) immediately after the name of a module. Similarly, arrays of imports between module instances can be linked together by specifying either an index or range specifier immediately after the name of an imported view.

```
DATABASE MODULE Plant;

Export     EngineerView: View
                           Unit1.EngineerView,
                           Unit2.EngineerView:View;
                         End;

           OperatorView: View
                           Unit1.OperatorView,
                           Unit2.OperatorView:View;
                         End;

           CorporateView: View
                            Expectations,
                            Management.Report:View;
                          End;

Import     Expectations: Production.Targets;

Type       Management; Unit;

Instance   Management; Unit1,Unit2:Unit;

Link       Management.Objectives  to  Expectations;
           Management.UnitView1   to  Unit1.OperatorView;
           Management.UnitView2   to  Unit2.OperatorView;

Performance  Response Time : 1-5 secs;

END. {Plant}
```

Fig. 4.14: Internal Links & Nested Views

The **constraint** section can specify structural restrictions on a configuration of modules, such as the fact that there must be an one-to-one correspondance between the 'Voltage' and 'Current' modules in Fig. 4.9, or that a module should not be linked to more than ten other modules. A notation based on first-order predicates together with some predefined functions is adequate for expressing such restrictions.

The **export** and **import** sections of a database module are similar to those of a data module. Since a database module does not declare any data objects, exported views can only be defined in terms of nested views:

i) Exported views of module instances declared within the database module. In Fig. 4.12, each of the modules 'Turbine1', 'Turbine2' and 'Turbine3' has two exported views 'EngineerView' and 'OperatorView', which are used to define the 'EngineerView' and 'OperatorView' of the encapsulating 'Unit' module in a nested manner. Similarly, the 'EngineerView' and 'OperatorView' of the 'Plant' in Fig. 4.14 are defined in a nested manner. The graphical representations for 'Unit' and 'Plant' are given in Fig. 4.10(c) & 4.10(d).

ii) Views imported from other modules. In Fig. 4.14, the view 'Expectations' is imported by 'Plant' but is also exported as part of the 'CorporateView'. Note that 'Management' is just another module and can be used for functions such as production scheduling, plant optimisation, etc. Since 'Management' is really administrative, it is given at most the capabilities of a plant operator for either 'Unit', but not the technical capabilities of a plant engineer. The 'CorporateView' can be used to receive management reports at the corporate headquarters.

Imported views used by the constituent modules of a database module must be linked to the internal modules that use them by means of explicit links, for example in Fig. 4.14, the 'Objectives' of 'Management' are explicitly linked to the 'Expectations' imported from the outside world.

Finally, **performance** is similar to that of a data module but the Response Time, Reliability and Consistency characteristics refer to the aggregate performance over all the constituent and imported modules of the database module. The performance characteristics of a database module can be checked for consistency with those of its constituent and imported modules. If nested database specifications are involved, these consistency checks may have to be applied recursively until data modules are eventually reached.

4.4 DCL Extension for Database Modification

The database configuration language (DCL) can be extended to allow modifications on an existing database configuration. Such modifications can be grouped and specified together in a **modify module.** A configuration manager similar to that described in [Kramer 85, Sloman 85] is assumed to process these modifications against the corresponding database specification in order to validate them, decide on the actual changes required, and initiate their execution on the underlying system. The approach is novel in that it can allow changes without shutting down the whole database or system as reconfiguration can be localised only to those components that are affected by a change.

Fig. 4.15 shows a modify module that expands 'AnalogueIn' of Fig. 4.9 by adding two new module instances, 'Voltage3' and 'Current3' respectively. The rest of the examples will also refer to this 'AnalogueIn' database. The name of a modify module and the name of the corresponding database module need not be the same, for example, the names 'AnalogueInChange' and 'AnalogueIn' are used throughout this section.

For the purpose of presentation, the specification of a database module may be viewed as a collection of attributes, an attribute being an item of specification. The 'AnalogueIn' database in Fig. 4.9 has two export attributes ('EngineerView' and 'OperatorView'), four type attributes ('Voltage', 'Current', 'ActivePower' and 'ReactivePower'), six instance attributes ('Voltage1', 'Voltage2', 'Current1', 'Current2', 'ActivePower' and 'ReactivePower'), a single constraint attribute ('COUNT(Voltage)=COUNT(Current)'), and three performance attributes ('Response Time', 'Reliability' and 'Consistency'). It is obvious now that an attribute is a pair of a "type" and a "value". The "type" of an attribute may be export, import, type, instance, constraint, link or performance. The "value" of an attribute is basically whatever can follow its "type" in the DCL specification of a database module.

```
MODIFY MODULE   AnalogueInChange;

insert instance Voltage3:Voltage;
                Current3:Current;

delete export   EngineerView;
                OperatorView;

insert export   EngineerView: View
                                Voltage1.EngineerView,
                                Voltage2.EngineerView,
                                Voltage3.EngineerView,
                                Current1.EngineerView,
                                Current2.EngineerView,
                                Current3.EngineerView,
                                ActivePower.EngineerView,
                                ReactivePower.EngineerView:View;
                              End;

                OperatorView: View
                                Voltage1.OperatorView,
                                Voltage2.OperatorView,
                                Voltage3.OperatorView,
                                Current1.OperatorView,
                                Current2.OperatorView,
                                Current3.OperatorView,
                                ActivePower.OperatorView,
                                ReactivePower.OperatorView:View;
                              End;
END. {AnalogueInChange}
```

Fig. 4.15: Modify Module With Six Modifications

A database module can be modified by applying a **delete**, **insert** or **replace** operation to one or more of the module attributes. A modification is thus a triplet comprising an operation, an attribute "type", and an attribute "value". A modify module is a collection of modifications. In Fig. 4.15 six modifications are specified: the creation of two new module instances called 'Voltage3' and 'Current3'; the deletion of the two exported views 'EngineerView' and 'OperatorView'; and the definition of two new exported views 'EngineerView' and 'OperatorView' which also allow access to the newly created modules 'Voltage3' and 'Current3'. The replace operation is included merely for specification convenience and is equivalent to a delete operation followed by an immediate insert operation. The modify module in Fig. 4.16 is equivalent to the modify module in Fig. 4.15.

```
MODIFY MODULE   AnalogueInChange;

insert instance Voltage3:Voltage;
                Current3:Current;

replace export  EngineerView by View
                                  Voltage1.EngineerView,
                                  Voltage2.EngineerView,
                                  Voltage3.EngineerView,
                                  Current1.EngineerView,
                                  Current2.EngineerView,
                                  Current3.EngineerView,
                                  ActivePower.EngineerView,
                                  ReactivePower.EngineerView:View;
                                End;

                OperatorView by View
                                  Voltage1.OperatorView,
                                  Voltage2.OperatorView,
                                  Voltage3.OperatorView,
                                  Current1.OperatorView,
                                  Current2.OperatorView,
                                  Current3.OperatorView,
                                  ActivePower.OperatorView,
                                  ReactivePower.OperatorView:View;
                                End;
END. {AnalogueInChange}
```

Fig. 4.16: Modify Module With Replace Operation

The semantics of the delete, insert and replace operations are as follows. An attribute is deleted only if no other attribute depends on it, for example, a type may be deleted if no instances of this type exist, or an instance may be deleted if none of its views is exported and it is not linked to other instances. An attribute is inserted only if it does not exist already and is semantically valid, for example, the type of an inserted instance must already exist, and a performance attribute is inserted only if it is satisfied by the current database configuration. An attribute replacement has the combined effect of a deletion followed by an immediate insertion as an atomic operation.

A replace operation has additional semantic repercussions when module instances and types are replaced. The **replace instance** operation may modify the name or the type or both the name and type of an existing module instance. Name modification does not incur any semantic problems. Type modification however is more difficult to handle: it implies that the new type of an old instance must also satisfy all the exports, imports and links of the old type, and also the integrity constraints

and performance of the surrounding database module. Similarly, a **replace type** operation implies **both** the modification of the existing type (not merely the definition of a new type) **and** the modification of all the existing instances of this type in the surrounding database module. Note that instance and type replacement has serious semantic repercussions and may be difficult to handle in practice. Nevertheless, it is worth having a separate replace operation because its expressive power allows modifications to be specified very concisely. In Fig. 4.17, the single 'ActivePower' instance within 'AnalogueIn' is replaced by its new version, but only the 'Current1' instance of 'Current' is replaced by its new version as 'Current2' still retains its old type 'Current'.

```
MODIFY MODULE  AnalogueInChange;
insert type        NewCurrent;
replace instance   Current1:Current by Current1:NewCurrent;
replace type       ActivePower by NewActivePower;
END. {AnalogueInChange}
```

Fig. 4.17: An Example of Instance & Type Replacements

Since database modules can be nested, modify modules should also be specified in a nested manner. In this way, "nested modifications" can be obtained and the structure of these modifications will reflect the nested structure of the database (cf. [Jackson 75]). Modifications are then modular and more likely to be correct. A facility is therefore required for invoking nested modifications. This can be done by extending the notation for type insertion and replacement to allow the construction of new types and the modification of existing ones by applying a modify module to an existing module type. Fig. 4.18 may be an alternative to the modifications in Fig. 4.17, assuming that 'ActivePower' and 'Current' are database modules. A modify module is essentially a transaction at the configuration level, so nested modify modules may be viewed as nested transactions at the configuration level.

```
MODIFY MODULE  AnalogueInChange;

insert type      NewCurrent          :  CurrentChange(Current);
                 {new definition}       {nested type construction}

replace instance Current1:Current by Current1:NewCurrent;

replace type     ActivePower         by ActivePowerChange(ActivePower);
                 {redefinition}         {nested type construction}

END. {AnalogueInChange}
```

Fig. 4.18: An Example of Type Construction & Modification

The configuration manager will have to perform such modifications in some specific order. For nested modifications, it will have to start from the innermost modules and proceed towards the outermost modules. Within a modify module, modifications can be performed in two ways. Firstly, modifications can be performed in the order they are listed. This is easier to handle because the configuration manager will perform the modifications in an explicit order externally defined. The validity of a modification will depend on the initial state of the configuration and on the modifications performed so far, for example, a module type is deleted if no more module instances of that type exist. Secondly, the configuration manager batches the modifications and determines the order of their execution. For example, insertions may be performed first, then deletions, and then replacements. Subsequently, for each kind of modification operation an additional order will have to be imposed, for example, deletion of links will have to precede deletion of instances, insertion of types will have to precede insertion of instances, etc. The order in which to perform the various modifications, can be derived from the operation dependencies between the modification operations (insert, delete, replace), and from the inter-attribute dependancies separately for each group of modification operations.

It is worth pointing out that all the above modifications require some sort of intelligent decision making by the configuration manager, especially with regard to the replacement of module instances and types. Moreover, these modifications will have to be performed atomically. Since the configuration manager is part of the system management described in section 2.4, it will have to operate both on the actual system and on the configuration database. It is the actual system rather

than the configuration database, that cannot easily ensure the atomicity of such modifications. Therefore, atomicity should ensure that both the actual system and the configuration database remain in a consistent state. Note that the DDL and DCL specifications will be part of the configuration database of the system. Note also that the description of a database module as a collection of attributes is similar to the semantic models described in [Hammer 81, Teichroew 80, Pirotte 77].

4.5 Chapter Summary

This chapter proposed a new and improved realtime database model that constitutes a synthesis of concepts and experiences from several computing disciplines. It provides the necessary concepts and a notation for the logical specification of a realtime database, allowing for the specification of data and database structures, operations, views, integrity constraints and performance characteristics. The data definition language (DDL) is used to specify individual database components, the data modules. The database configuration language (DCL) is subsequently used to specify a database module as interconnected instances of data modules and perhaps of other database modules. The DCL can also be extended to specify database modifications. The approach is novel in that it can allow modifications in the form of dynamic reconfiguration without shutting down the whole database or system. The formal definitions for the DDL, the DCL, and the DCL extension can be found in the Appendix. The use of the proposed model is demonstrated further by a complete example which is presented in the next chapter.

CHAPTER 5

A COMPLETE EXAMPLE

A complete example is now used to demonstrate how the realtime database model presented in the previous chapter can be applied to real world situations. The example concerns a simplified patient monitoring system and has often been used for demonstration purposes in the computer literature, for example, [Stevens 74, Magee 84, Kramer 85]. The details of this example can be formulated as following:

"A patient monitoring system is required for a hospital. Each patient is monitored by a number of sensors which measure pulse, blood pressure and temperature. The system reads these measurements on a periodic basis. For each patient, safe ranges of each measurement are specified. If a measurement falls outside the safe range, or if a sensor fails, an alarm is raised at once. The database for a single patient consists of the current status for each sensor, status and alarms summary from all sensors, and status history for the last week".

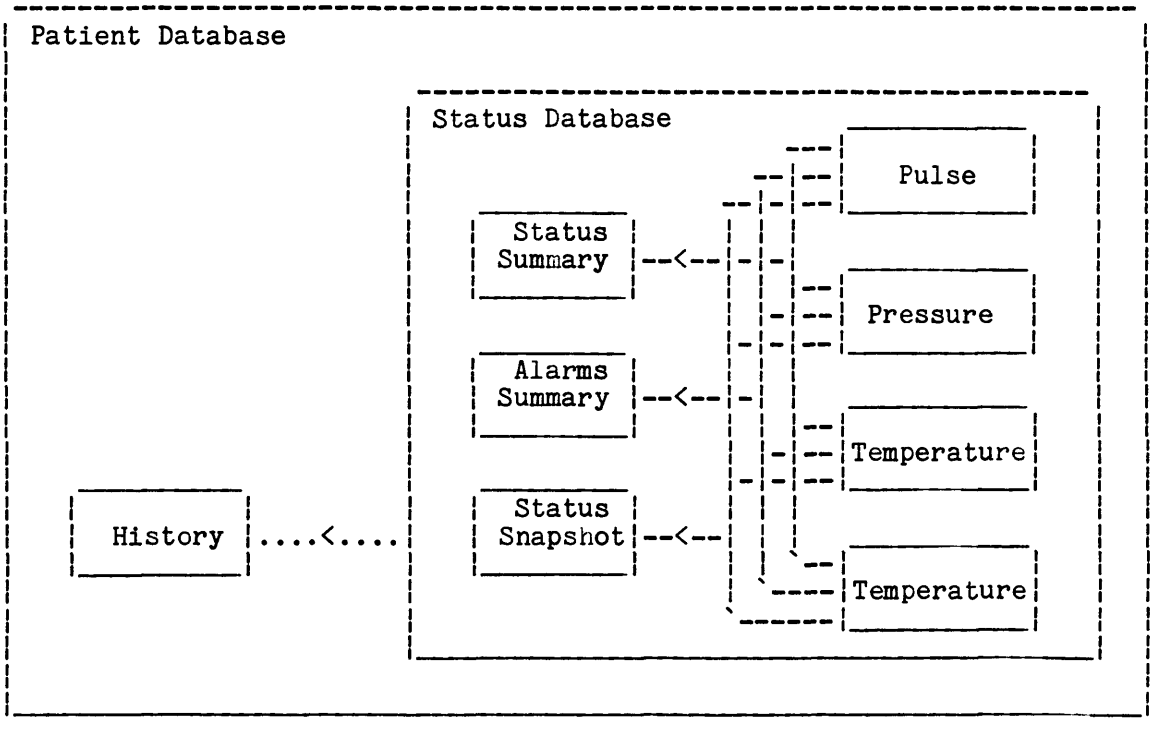


Fig. 5.1: The data & database modules related to a single patient.

The patient database can be modelled as shown in Fig. 5.1. A rectangle represents a data or database module; a solid arrow means that the module pointed to by the arrow imports a view from the module at the tail of the arrow; and a dotted arrow indicates that the data within the module pointed to by the arrow is produced after some processing of data originating in the module at the tail of the arrow.

The 'Pulse', 'Pressure' and two 'Temperature' data modules correspond to the sensors monitoring a patient. Each of these sensor data modules contains the parameters for a sensor: physical sensor address, current value, scan period, upper and lower limits, range and fail alarms. It also defines several views on the stored parameters:

* 'ValueUpdate', 'RangeAlarmUpdate' and 'FailAlarmUpdate' that are used by processing modules (not shown in Fig. 5.1) to periodically update the current value, raise the range alarm if the current value is not within limits, and notify the failure of the sensor by raising the fail alarm.

* 'StatusSubset' used by the 'StatusSummary' data module to maintain local copies of the current value and range alarm from all the sensor data modules.

* 'AlarmsSubset' used by the 'AlarmsSummary' data module to maintain local copies of the range and fail alarms from all the sensor data modules.

* 'SnapshotData' used by the 'StatusSnapshot' data module to produce a snapshot of the current values, range and fail alarms from all the sensor data modules.

* 'ManMachineInterface' used by a processing module (not shown in Fig. 5.1) to provide a man-machine interface, say to a nurse.

The 'Status' database module corresponds to the current state of a patient. It contains seven instances of data modules and defines two nested views from these instances:

* 'ManMachineInterface' used by the man-machine interface processing module to provide access to the sensor and summary data modules.

* 'HistoryCollection' allowing access to the 'StatusSnapshot' data module and used by a historian, an implied processing module in Fig. 5.1. The historian obtains the current status of a patient periodically, processes it, and, at the end of the day, records past average, maximum, and minimum values together with timestamps and alarm conditions in the 'History' data module.

The 'Patient' database module contains one instance from each of the 'Status' and 'History' modules, and defines a 'ManMachineInterface' view that is used by the man-machine interface processing module to provide access to the current status and history of a patient. Note that there must exist an one-to-one correspondance between the 'Status' and 'History' modules.

The specification of the patient monitoring system proceeds as follows. All the data types required for the example are defined first. Then, the 'Pulse', 'StatusSummary', 'AlarmsSummary', 'StatusSnapshot' and 'History' data modules are specified in the DDL. The 'Pressure' and 'Temperature' data modules are omitted because they are similar to 'Pulse'. The 'Status' and 'Patient' database modules are subsequently specified in the DCL. Some modifications that can be applied to the patient database are also specified: the addition of personal information for a patient, the addition of a skin resistance sensor, and the replacement of one of the temperature sensors by another version. Finally, some further modifications that may be applied to the patient database are specified in order to show how data modules can be used to model the background and foreground data of display frames.

5.1 Data Types

```
Measurement : Integer; {in system-wide units}

TimeInterval: Integer, values 0..1209600;  {in secs, up to  2 weeks}

TimeStamp   : Integer, values 0..86400;    {in secs, up to 24 hours}

ValueAndLimits: Record
                  CurrentValue, HighLimit, LowLimit: Measurement;
                End;

ValueAndAlarms: Record
                  CurrentValue: Measurement;
                  RangeAlarm, FailAlarm: Boolean;
                End;

AllDetailsDataType: Record
                      SensorAddress: Integer;
                      CurrentValue: Measurement;
                      ScanPeriod: TimeInterval;
                      UpperLimit, LowerLimit: Measurement;
                      RangeAlarm, FailAlarm, CombinedAlarm: Boolean;
                    End;

AllStatusSummaryDataType: Array [1..3] of
                            Record Value:Measurement; Alarm:Boolean End;

AllAlarmsSummaryDataType: Array [1..3] of
                            Record RangeAlarm, FailAlarm:Boolean End;

CurrentSnapshot: Array [1..4] of ValueAndAlarms;

Day: (Mon,Tues,Wed,Thus,Fri,Sat,Sun);

HistoryRecord: Record
                 Maximum:Record
                           Value:Measurement;
                           Time :TimeStamp;
                           Alarm:Boolean;
                         End;
                 Minimum:Record
                           Value:Measurement;
                           Time :TimeStamp;
                           Alarm:Boolean;
                         End;
                 Average:Measurement;
               End;

HistorySubEntry: Record
                   SensorFailure:Boolean, initially true;
                   Case SensorFailure of
                        false: (Record:HistoryRecord);
                        true : ( );
                   End;
                 End;

HistoryEntry: Array [1..3] of HistorySubEntry;

HistoryDataType: Array [Day] of HistoryEntry;
```

5.2 Data Modules

```
DATA MODULE Pulse;

Export ValueUpdate: View
                      SensorAddress, ScanPeriod:Read; Value:Write;
                    End;

       RangeAlarmUpdate: View
                           ScanPeriod, CheckData:Read; RangeAlarm:Write;
                         End;

       FailAlarmUpdate : View
                           SensorAddress:Read; FailAlarm:Write;
                         End;

       StatusSubset: View
                       Value:Periodic 80 msec; RangeAlarm:OnChange;
                     End;

       AlarmsSubset: View
                       RangeAlarm, FailAlarm:OnChange;
                     End;

       SnapshotData:Read;

       ManMachineInterface: View
                              SensorAddress, ScanPeriod, UpperLimit,
                              LowerLimit, RangeAlarm, FailAlarm:Write;
                              AllPulseData:Read;
                            End;
Type   Measurement;
       TimeInterval;
       ValueAndLimits;
       ValueAndAlarms;
       AllDetailsDataType;

Data   SensorAddress          : Integer;
       Value                  : Measurement;
       ScanPeriod             : TimeInterval, values 100..500;
       UpperLimit             : Measurement, initially 100;
       LowerLimit             : Measurement, initially 50;
       RangeAlarm, FailAlarm  : Boolean, initially false;
       CombinedAlarm: Boolean = RangeAlarm or FailAlarm;

       CheckData    : ValueAndLimits <Value,UpperLimit,LowerLimit>;
       SnapshotData : ValueAndAlarms <Value,RangeAlarm,FailAlarm>;
       AllPulseData : AllDetailsDataType
                      <SensorAddress,Value,ScanPeriod,UpperLimit,
                       LowerLimit,RangeAlarm,FailAlarm,CombinedAlarm>;

Constraint   UpperLimit >= LowerLimit;
             (SensorAddress mod 2) = 0;  {even addresses only}

Performance  Response Time : 100 msec;
             Reliability   : warm backup;
             Consistency   : medium;

END. {Pulse}
```

```
DATA MODULE StatusSummary;

Export  AllStatusSummary:Read;

Import  PulseView: Pulse.StatusSubset;
        PressView: Pressure.StatusSubset;
        Temp1View,
        Temp2View: Temperature.StatusSubset;

Type    Measurement;
        AllStatusSummaryDataType;

Data    PulseValue: Measurement <-- PulseView.Value      periodic;
        PulseAlarm: Boolean     <-- PulseView.RangeAlarm onchange;

        PressValue: Measurement <-- PressView.Value      periodic;
        PressAlarm: Boolean     <-- PressView.RangeAlarm onchange;

        Temp1Value: Measurement <-- Temp1View.Value      periodic;
        Temp1Alarm: Boolean     <-- Temp1View.RangeAlarm onchange;

        Temp2Value: Measurement <-- Temp2View.Value      periodic;
        Temp2Alarm: Boolean     <-- Temp2View.RangeAlarm onchange;

        AverageTemp: Measurement = (Temp1Value+Temp2Value) div 2;
        AggregateTempAlarm: Boolean = Temp1Alarm or Temp2Alarm;

        AllStatusSummary: AllStatusSummaryDatatype
                          <PulseValue,PulseAlarm,
                           PressValue,PressAlarm,
                           AverageTemp,AggregateTempAlarm>;

Performance  Response Time : 200 msec;
             Reliability   : cold backup;
             Consistency   : medium;

END. {StatusSummary}
```

```
DATA MODULE AlarmsSummary;

Export      AllAlarmsSummary:Read;

Import      PulseView: Pulse.AlarmsSubset;
            PressView: Pressure.AlarmsSubset;
            Temp1View,
            Temp2View: Temperature.AlarmsSubset;

Type        AllAlarmsSummaryDataType;

Data        PulseRangeAlarm: Boolean <-- PulseView.RangeAlarm onchange;
            PulseFailAlarm : Boolean <-- PulseView.FailAlarm  onchange;
            PressRangeAlarm: Boolean <-- PressView.RangeAlarm onchange;
            PressFailAlarm : Boolean <-- PressView.FailAlarm  onchange;
            Temp1RangeAlarm: Boolean <-- Temp1View.RangeAlarm onchange;
            Temp1FailAlarm : Boolean <-- Temp1View.FailAlarm  onchange;
            Temp2RangeAlarm: Boolean <-- Temp2View.RangeAlarm onchange;
            Temp2FailAlarm : Boolean <-- Temp2View.FailAlarm  onchange;

            TempRangeAlarm: Boolean = Temp1RangeAlarm or Temp2RangeAlarm;
            TempFailAlarm : Boolean = Temp1FailAlarm  or Temp2FailAlarm;

            AllAlarmsSummary: AllAlarmsSummaryDataType
                              <PulseRangeAlarm,PulseFailAlarm,
                               PressRangeAlarm,PressFailAlarm,
                               TempRangeAlarm, TempFailAlarm>;

Performance Response Time : 200 msec;
            Reliability   : hot backup;
            Consistency   : medium;

END. {AlarmsSummary}
```

```
DATA MODULE StatusSnapshot;

Export      ReadCurrentSnapshot:Transaction;

Import      PulseView: Pulse.SnapshotData;
            PressView: Pressure.SnapshotData;
            Temp1View,
            Temp2View: Temperature.SnapshotData;

Type        CurrentSnapshot;

Transaction ReadCurrentSnapshot (out Snapshot:CurrentSnapshot);
              Begin
                PulseView.Read(SnapshotData,Snapshot[1]);
                PressView.Read(SnapshotData,Snapshot[2]);
                Temp1View.Read(SnapshotData,Snapshot[3]);
                Temp2View.Read(SnapshotData,Snapshot[4]);
              End;

Performance Response Time : 1-5 sec;
            Reliability   : cold backup;
            Consistency   : strong;

END. {StatusSnapshot}



DATA MODULE History;

Export  ManMachineInterface: View
                               ScanPeriod:Read,Write; History: Read;
                             End;

        HistoryUpdate: View
                         ScanPeriod:Read; UpdateEntry:Transaction;
                       End;

Type    TimeInterval; HistoryDataType; HistoryEntry; Day;

Data    ScanPeriod: TimeInterval, values 1..15; {every 1-15 sec}
        History   : HistoryDataType;

Transaction  UpdateEntry (in Index:Day; DailyEntry:HistoryEntry);
               Begin
                 History[Index]:=DailyEntry;
               End;

Performance  Response Time : 5-30 sec;
             Reliability   : warm backup;
             Consistency   : strong;

END. {History}
```

5.3 Database Modules

```
DATABASE MODULE Status;

Export   HistoryCollection: View
                              StatusSnapshot.ReadCurrentSnapshot:View;
                            End;

         ManMachineInterface: View
                                Pulse.ManMachineInterface,
                                Press.ManMachineInterface,
                                Temp1.ManMachineInterface,
                                Temp2.ManMachineInterface,
                                StatusSummary.AllStatusSummary,
                                AlarmsSummary.AllAlarmsSummary:View;
                              End;

Type      Pulse;
          Pressure;
          Temperature;
          StatusSummary;
          AlarmsSummary;
          StatusSnapshot;

Instance  Pulse;
          Press: Pressure;
          Temp1, Temp2: Temperature;
          StatusSummary;
          AlarmsSummary;
          StatusSnapshot;

Link      StatusSummary.PulseView  to  Pulse.StatusSubset;
          StatusSummary.PressView  to  Press.StatusSubset;
          StatusSummary.Temp1View  to  Temp1.StatusSubset;
          StatusSummary.Temp2View  to  Temp2.StatusSubset;

          AlarmsSummary.PulseView  to  Pulse.AlarmsSubset;
          AlarmsSummary.PressView  to  Press.AlarmsSubset;
          AlarmsSummary.Temp1View  to  Temp1.AlarmsSubset;
          AlarmsSummary.Temp2View  to  Temp2.AlarmsSubset;

          StatusSnapshot.PulseView to  Pulse.SnapshotData;
          StatusSnapshot.PressView to  Press.SnapshotData;
          StatusSnapshot.Temp1View to  Temp1.SnapshotData;
          StatusSnapshot.Temp2View to  Temp2.SnapshotData;

Performance  Response Time : 0.5-1.0 sec;
             Reliability   : warm backup;
             Consistency   : medium;

END. {Status}
```

```
DATABASE MODULE Patient;

Export      ManMachineInterface: View
                                   Status.ManMachineInterface,
                                   History.ManMachineInterface:View;
                                 End;
Type        Status; History;

Instance    Status; History;

Constraint  COUNT(Status)=COUNT(History);

Performance Response Time : 1-30 sec;
            Reliability   : warm backup;
            Consistency   : medium;

END. {Patient}
```

5.4 Modifications

```
DATA MODULE PersonalInfo;

Export  Name: OnChange;

        Details: View
                   Name,Age:Read,Write;
                 End;

Data    Name: String(12);

        Age : Integer, values 0..100;

END. {PersonalInfo}


MODIFY MODULE PatientChange;

delete export    ManMachineInterface;

insert type      PersonalInfo;

insert instance  Personal:PersonalInfo;

insert type      NewStatus:StatusChange(Status); {nested modification}

replace instance Status by Status:NewStatus;

insert export    ManMachineInterface:View
                                       Personal.Details,
                                       Status.ManMachineInterface,
                                       History.ManMachineInterface:View;
                                     End;
END. {PatientChange}
```

```
MODIFY MODULE StatusChange;

insert type      SkinResistance;

insert instance  Skin:SkinResistance;

insert type      NewTemperature;

replace instance Temp1 by Temp1:NewTemperature;

replace export  ManMachineInterface:View
                                     Pulse.ManMachineInterface,
                                     Press.ManMachineInterface,
                                     Temp1.ManMachineInterface,
                                     Temp2.ManMachineInterface,
                                     Skin.ManMachineInterface,
                                     StatusSummary.AllAlarmsSummary,
                                     AlarmsSummary.AllAlarmsSummary:View;
                                    End;

END. {StatusChange}
```

The 'SkinResistance' and 'NewTemperature' data modules are similar to 'Pulse'. Note also that after these modifications are carried out, neither summary information nor history will be collected from the skin resistance sensor. If such information is also required, then additional data and database modules will have to be (re)configured.

5.5 Further Modifications

This section shows how display frames may be added to the patient database specified so far. For simplicity, only a single display frame that contains the name and pulse rate of a patient is specified. This pulse display frame appears pictorially in Fig. 5.2. It is intended for a simple graphics interface that can display up to five items, can draw rectangles and circles, and can print string, integer, real and boolean tokens with user-selected background and foreground colours.

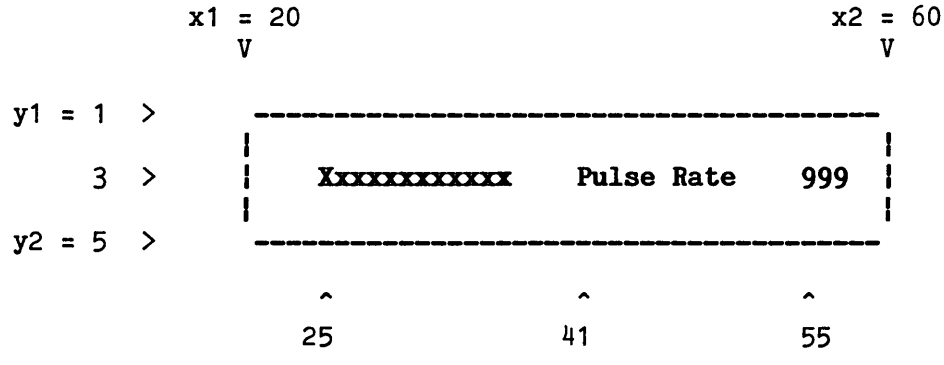


Fig. 5.2: A Display Frame & Its Coordinates

```
display: Array [1..5] of icon;

icon: Record
        kind: (null,full);
        Case kind of
          null: ( );
          full: (it:item);
        End;
      End;

item: Record
        kind: (rectangle,circle,text,numeric,floating,binary);
        Case kind of
          rectangle: (x1, x2:xcoordinate; y1, y2:ycoordinate);
          circle   : (c:point;  r:radius);
          text     : (s:string(80);  p:point;  d:depth);
          numeric  : (i:integer;     p:point;  d:depth);
          floating : (r:real;        p:point;  d:depth);
          binary   : (b:boolean;     p:point;  d:depth);
        End;
        bcolour: colour;  {background or  inside colour}
        fcolour: colour;  {foreground or outline colour}
     End;

xcoordinate : integer, values 1..80;
ycoordinate : integer, values 1..24;

point : Record x:xcoordinate; y:ycoordinate End;

radius: integer, values 1..10;
depth : integer, values 1..80; {token size in characters}

colour: (nocolour,yellow,orange,red,green,blue,black,white);



DATA MODULE  PulseDisplay;

Export Background, Foreground, AllDisplay: Read;

Import Personal: PersonalInfo.Name;
       Pulse   : Pulse.StatusSubset;

Type   Measurement, icon, display;

Data   PatientName: String(12)  <-- Personal.Name onchange;
       PulseValue : Measurement <-- Pulse.Value   periodic;

       icon1: icon <full,<rectangle,<20,60,1,5>,nocolour,white>>;
       icon2: icon <full,<text,<PatientName,<25,3>,12>,nocolour,yellow>>;
       icon3: icon <full,<text,<"Pulse Rate",<41,3>,10>,nocolour,green>>;
       icon4: icon <full,<numeric,<PulseValue,<55,3>,3>,green,yellow>>;

       Background: display <icon1,icon3,<null>,<null>,<null>>;
       Foreground: display <icon2,icon4,<null>,<null>,<null>>;
       AllDisplay: display <icon1,icon2,icon3,icon4,<null>>;

END. {PulseDisplay}
```

Foreground data need not always be stored as copy data objects but can reside in other data modules, for example, the detailed parameters of a temperature sensor. In that case, the display frame will be represented by a data module with one or more transactions. A transaction is used in order to access the required data objects dynamically via imported views, and then assemble them to form the foreground data.

5.6 Chapter Summary

This chapter demonstrated the use of the realtime database model in real world situations by means of an example, a simplified patient monitoring system. All the data types required for the specification of the patient database were defined first. Then, five data modules were specified in the DDL: 'Pulse', 'StatusSummary', 'AlarmsSummary', 'StatusSnapshot' and 'History'. Two database modules were subsequently specified in the DCL: 'Status' and 'Patient', with 'Status' being nested within 'Patient'. Three modifications that can be applied to this patient database were also specified: the addition of personal information for a patient, the addition of a skin resistance sensor, and the replacement of one of the temperature sensors by another version. Some further modifications were finally specified in order to show how a data module can be used to model the backgound and foregound data of display frames. A prototype implementation of this patient monitoring system has been developed for demonstration purposes. The implementation is in terms of the component architecture that is described in the next chapter.

CHAPTER 6

IMPLEMENTATION

This chapter presents a component architecture, essentially a refinement of the realtime database model, that can be used to implement the overall functionality of a data module. A number of run-time mechanisms are included in order to support the various performance characteristics provided by the model, i.e. response times, levels of consistency and reliability backups. These mechanisms comprise pointer access for fast response times, concurrency control, recovery from communication failures, recovery from physical node failures, and atomic transactions. Some of these mechanisms and all the functionality of a data module were implemented and tested in a prototype implementation of the patient monitoring system specified in chapter 5. A number of issues related to the implementation cost of these run-time mechanisms are also discussed.

6.1 Component Architecture

6.1.1 Basic Components

The architecture comprises four basic components with well-defined interfaces and interactions between them. These components are:

* A **Pm** module corresponds to a processing module. It performs operations such as reading sensors and storing their values in the database, implementing control algorithms, generating histories and logs, providing man-machine interfaces, etc.

* A **Dm** module contains the data objects defined by a data module. Data objects may reside in main memory, disk or bubble storage, depending on the response times required. A Dm provides for all kinds of data objects, value and integrity constraints, and read, write, periodic and onchange actions. It also implements transactions on the data objects which it contains, but not transactions on data objects in other Dm's. A Dm provides mechanisms for controlling the concurrent transactions implemented by Tm's.

* A **Tm** module implements a single transaction across a number of Dm's. Thus, it invokes all the necessary actions on multiple Dm's in order to implement an atomic transaction.

* An **Im** module interfaces a Pm to Tm's and Dm's. It maps the operations and data objects as seen by a Pm to those offered by Tm's and Dm's. It is responsible for data type conversion, retransmission of action invocations, handling of timeouts, etc. It should be considered part of a Pm and so normally resides on the same physical node as the Pm. An Im is similar to the client-stub of [Nelson 81, Birrell 84] and can be generated automatically from Pm and Dm descriptions.

Fig. 6.1 shows the inter-relationship between these modules. An Im cannot be shared by Pm's but a Pm may access several Im's. Tm's and Dm's act as servers supporting multiple clients either directly or via Im's. A Tm can access several Dm's. A Dm can update secondary copies of data objects in other Dm's. Note that the diagram of Fig. 6.1 is very similar to the graphical representations associated with semantic models and with the entity-relationship model in particular.

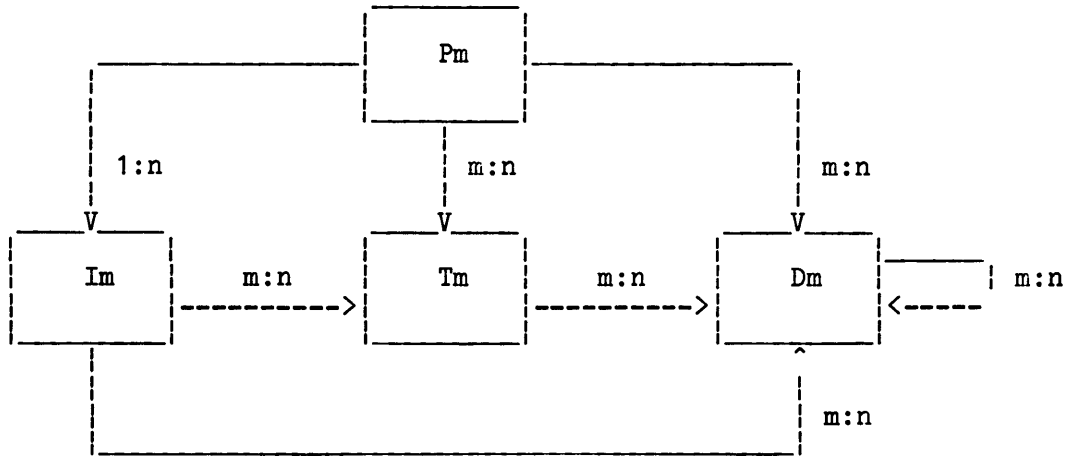


Fig. 6.1: Inter-Relationships Between Components

All modules are assumed to have a single thread of execution. When a Pm invokes an action or transaction, it waits for a reply to be returned. A Tm does not invoke actions in parallel. A Dm serves actions serially. Action invocations are queued at the Dm interface until they are serviced. If more concurrency is required, a Dm will have to be implemented by several server modules that will handle the concurrent action invocations on the Dm, as suggested in [Liskov 83, Allchin 83].

The achitecture of Fig. 6.1 follows the same philosophy of functional components as the NBS architecture [NBS 82]. However, the overall approach taken in this thesis is different from the NBS approach in that it separates the storage and retrieval of data (realtime database) from the storage and retrieval of meta data (configuration database). The architecture of Fig. 6.1 is also similar to the ANSI architecture [ANSI 78]. An Im has some similarity to an external schema, all the Dm's seem to correspond to the conceptual schema, and the Dm's could be extended to access internal modules (internal schemas) for handling disk or bubble storage. As compared to both the NBS and the ANSI architectures, the architecture of Fig. 6.1 is confined to the run-time components required for the implementation of a realtime database and does not include the components required for the management of a realtime database, such as data dictionary, dynamic reconfiguration tools, etc.

6.1.2 Intermodule Communication

The communication between the modules of Fig. 6.1 has the form of client-server interaction, so it can be cast either as request-reply message passing or as (remote) procedure calls [Andrews 83, Birrell 84, Nelson 81]. For remote accesses, message passing and remote procedure calls have comparable response times, but for local accesses, procedure calls are usually more efficient. However, the periodic and onchange actions are best suited to asynchronous, unidirectional and unreliable message passing as discussed extensively in [Gueth 85a & 85b]. Using request-reply message passing or (remote) procedure calls for updating secondary copies of data objects would add significant run-time overheads on the implementation of periodic and onchange actions.

For efficiency reasons, a Pm is also allowed to access a base or copy data object within a Dm on the same physical node via pointer access, thus bypassing the proper Dm interface. This pointer is given by the Dm to the Pm via a special get-address action.

No consideration is given in this thesis to finding the best mechanisms for accessing Dm's from Pm's. There are three alternatives [Gallagher 84]: explicit procedure calls where a Pm calls external procedures which in turn will access the Dm's; implicit procedure calls where a Pm is interspersed with calls to Dm's, which will be processed by a preprocessor into external procedures; and native language syntax where

the language (and hence the compiler) for Pm's is extended to handle database concepts directly. The prototype of section 6.5 uses message passing which is similar to explicit procedure calls. The CUTLASS system [Bransby 82, Barson 82], the CERN SPS system [Crowley-Milling 74 & 79], and the Industrial RealTime Basic [Kahn 83] use native language syntax.

6.2 Performance Characteristics

The purpose of this section is to enumerate the run-time mechanisms that will have to be incorporated in the components of the architecture in order to support the performance characteristics provided by the model, i.e. response times, levels of consistency and reliability backups. All these mechanisms are described in detail in section 6.3.

6.2.1 Response Times

The run-time response times depend on the underlying system which may provide priority or deadline scheduling of tasks and fast communication between physical nodes. However, run-time response times are beyond the control of Dm's and Tm's. From an invoker's point of view, response times can be met by associating a timeout with each action or transaction invocation. Section 6.3.1 discusses how data objects in main memory and pointer access can be used to achieve fast response times.

The response times of Dm's and Tm's can be calculated at specification, compile, or even configuration time by summing up the longest execution paths ever to be taken. Execution paths are likely to be represented as sequences of statements in a programming language. Deadline calculations can be used to evaluate these paths, using a table with the execution times for each kind of statement. A different table should be used for each different target machine. Communication delays between modules on different physical nodes should also be taken into consideration.

6.2.2 Levels of Consistency

Weak: For efficiency reasons, pointer access to base and copy data objects is allowed for Pm's on the same physical node as the Dm. This is described in section 6.3.1. Pointer access compromises the integrity of a Dm, hence the term "weak" consistency.

Medium: Pointer access is allowed for the base and copy data objects that will be read only, while all write actions go via the proper Dm interface. Locks will have to be used for write actions that are part of transactions. This is described in section 6.3.2. The mechanisms for ensuring medium consistency in the presence of communication and physical node failures are described in sections 6.3.3 to 6.3.9. Since a write action on multiple data objects cannot be done atomically, it is likely that Pm's using pointer access may read some new and some old values, and the integrity constraints on the Dm may be temporarily violated.

Strong: All the actions are invoked via the proper interface provided by a Dm. Pointer access is not allowed. Locks will have to be used for both read and write actions that are part of transactions. A detailed description of the mechanisms required for strong consistency is given in section 6.3.2. The mechanisms for ensuring strong consistency in the presence of communication and physical node failures are described in sections 6.3.3 to 6.3.9.

6.2.3 Levels of Reliability

None: A Dm is implemented on its own. If a Dm fails, actions and transactions requested from the Dm will never be serviced.

Cold: A Dm is associated with a backup Dm' on another physical node. If the Dm fails, the Dm' detects the failure and takes over as a newly initialised module. In case of Dm failure, all the values of its data objects and state information will be lost. The mechanisms for implementing cold reliability are a subset of the mechanisms for implementing warm reliability.

Warm: A Dm is associated with a backup Dm' on another physical node and the values of the data objects in Dm are periodically copied to the Dm'. If the Dm fails, the Dm' will detect the failure and take over. The values of the data objects in Dm' will be consistent, but may not represent the most recent values in Dm. The mechanisms for implementing warm reliability are a subset of the mechanisms for implementing hot reliability.

Hot : A Dm is associated with a backup Dm' on another physical node. The data objects in Dm' are always a mirrored image of the data objects in Dm. If the Dm fails, the Dm' will detect the failure and take over. The mechanisms for implementing hot reliability are described in sections 6.3.5 and 6.3.6.

6.3 Run-time Mechanisms

This section describes the run-time mechanisms required for supporing the performance characteristics provided by the model. These mechanisms are well-known and have been documented extensively in the literature. They include pointer access for fast response times, concurrency control, recovery from communication failures, backup modules and a two phase commit protocol (2PC) for recovering from physical node failures, provision of atomic transactions, and transfers of large data objects.

6.3.1 Fast Response Times

Fast response times can be achieved by implementing the data objects of a Dm in main memory and by allowing pointer access to base and copy data objects for Pm's on the same physical node as the Dm. A pointer to a data object is obtained by means of a get-address action on the Dm. Use of pointers can satisfy even applications with extreme realtime constraints, but compromises the integrity, data independence and modularity provided by a Dm. Thus, use of pointers makes modifications very difficult and is not amenable to dynamic reconfiguration.

6.3.2 Concurrency Control

Concurrency control is required for supporting medium and strong consistency. Since a Dm serialises the execution of all actions, individual actions require no concurrency control. The problem lies with transactions (Tm's) that can invoke several actions. If transactions are serialised, then conflict never arises. However, substantial performance can be gained by allowing parallelism. Concurrency control ensures that the data objects within a Dm remain consistent in the presence of concurrent transactions. The solution to concurrency control involves a **synchronisation** mechanism that provides serialisability, and a **recovery** mechanism that provides restoration when a transaction is aborted.

The main synchronisation mechanisms are locking and timestamps [Kohler 81, Bernstein 81]. Although the model is amenable to both mechanisms, **locking** was chosen because it is straightforward and not expensive to implement. The locking mechanism is based on well-formed transactions that use two phase locking [Eswaran 76, Gray 78]. A transaction is well-formed if it it locks each data object before accessing it, and unlocks each data object before it completes. Two phase locking requires that all locks are acquired during an initial (growing) phase and are released during a final (shrinking) phase. No locks are released during the initial phase and no locks are acquired during the final phase. Once a lock is released no other lock can be acquired. To avoid cascading aborts, all locks should be held until a transaction commits or aborts.

A Dm will have to associate a lock with each data object and to provide lock, commit and abort actions. An unlock action is not needed since this can be done implicitly by commit and abort actions. A further distinction can be made between read and write locks in order to increase concurrency [Bernstein 81, Liskov 83]. If more concurrency is required, additional semantic information about the actions and transactions on a Dm can be exploited as suggested in [Allchin 83].

A transaction identifier **Tid** must exist for each transaction and be used for concurrency control. This Tid must be supplied as a parameter to all the actions which are invoked by the transaction. Receiving a request for a lock action with an unknown Tid will automatically create a new transaction within the Dm. Thus there is no need for a special action that starts a new transaction. The responsibility for generating Tid's rests with Tm's. A Tid can be obtained by concatenating the TmId of a Tm instance with a TransId which identifies the current invocation of the transaction. TmId must be globally unique among Tm instances and this must be ensured at Tm instantiation time. TransId can be a counter which is incremented whenever the transaction of a Tm is invoked.

For recovery purposes, **current** and **shadow** versions are maintained for each data object [Moss 81, Allchin 83]. All writes are done tentatively on current versions. If a transaction commits, the current versions of the data objects written by the transaction are copied to their shadow versions. If a transaction aborts, the shadow versions are used to restore the initial values of the current versions.

Locking can cause deadlock, however. Given the inherent difficulties and cost to avoid, prevent or detect deadlock within a distributed system, the implementation uses a simple timeout solution as suggested also in [Borr 81, Lampson 81, Liskov 83, Allchin 83]. A timeout is associated with each lock and consequently with each data object. The same timeout period can be used for all the data objects in a Dm. A lock is broken when its timeout expires. When a lock is broken, the entire transaction is aborted by the Dm and other locks may be released as well. Use of lock timeouts means that a Dm may unilaterally abort a transaction. Thus a 2PC must be used by Tm's to coordinate the commitment of a transaction across a number of Dm's. This 2PC is needed only for concurrency control and has also been adopted in [Moss 81, Bernstein 81].

6.3.3 Communication Failures

Invocation of remote actions and transactions must be able to cope with communication failures. Messages containing requests and replies of action (or transaction) invocations on a Dm can be delayed, lost, corrupted and discarded. It is assumed that no permanent partitioning or failure of the underlying communication system is possible. This can be ensured by means of redundant communication paths. It is also assumed that messages cannot arrive out of order. A monotonically increasing timestamp can be appended to all messages so that out-of-order messages can be filtered out by the receiver. Such a timestamp can be made by concatenating the sender's physical node identifier with the local time.

Delayed, lost and discarded messages can be overcome by retransmission of an action invocation request until a timeout expires or a retry limit is exceeded. Retransmission can cause duplicate requests which must be filtered out by the receiving Dm. For this reason, each action invocation is assigned a unique action identifier **ActionId.** The invoker (Pm or its Im, or Tm) periodically retransmits a request using the same ActionId as the original action invocation. A Dm receives a request and checks whether an action with such an ActionId has been carried out already. If it has, the Dm returns the saved reply. Otherwise, the Dm carries out the action and then returns the reply. In this way, duplicate requests are filtered out and the action is done only once. Retransmission of requests caters for both lost requests and lost replies. Receipt of an action invocation with a higher ActionId can serve as an acknowledgement for the last action invocation, so the

request and reply of the last action invocation can be forgotten. For Tm's, an ActionId can be obtained by concatenating the Tid described in section 6.3.2, with a transaction-relative identifier which is unique and monotonically increasing for each action invoked by the Tm.

Each Dm will have to keep an **action list** with one entry for each known client (Pm or Tm). Each entry will contain the Tid for a client, the ActionId of the last action invocation, and the reply of the last action invocation. Receipt of an action invocation from an unknown Tid automatically creates a new entry in the action list. If the size of an action list becomes a problem, entries in the list may be discarded after a large time interval. Considering that read, lock and abort actions (during the first phase of the 2PC) are idempotent and can be easily redone, they need not be recorded in the action list.

Similarly, to overcome communication failures between a Pm (or its Im) and a Tm, requests for transaction invocations must be retransmitted. A Pm (or its Im) will have to supply a **TransactionId** (similar to an ActionId) which uniquely identifies the invocation of the transaction, so that duplicate requests can be filtered out by the receiving Tm. Each Tm will have to maintain a **transaction list** (similar to an action list) that will be used to record previous transaction invocations, recognise duplicate requests, and ensure that a transaction is executed only once.

6.3.4 Invocation Semantics

The semantics proposed for the invocation of actions (or transactions) on a Dm or for the invocation of transactions on a Tm can be stated as following. If a reply is returned, the action or transaction was either done or not done at all. No guarantee is made that a reply will be returned eventually. If no reply is returned, the invoker will not know whether the action or transaction was done or not. According to section 6.3.3 however, using a new ActionId for each action (transaction) invocation on a Dm and maintaining an action list in a Dm can guarantee that the action was not done more than once. Similarly, using a new TransactionId for each transaction invocation on a Tm and maintaining a transaction list in a Tm can guarantee that the transaction was not done more than once. This semantics is similar to the remote procedure call semantics proposed in [Birrell 84].

The reply returned to an invoker can be "Done", "NotDone" or "Unknown". "Done" means that the action or transaction was carried out correctly and exactly once. "NotDone" means that the action or transaction was rejected and not done at all. This may be accompanied by a reason for the rejection, such as unknown data object, integrity constraint violation, etc. "Unknown" means that the action or transaction was executed zero or one times, but the invoker will not know which. The invoker can retry however in order to find out the outcome of the action or transaction, or to carry it out eventually. "Unknown" may also be accompanied by a reason, for example, timeout expired, retry limit exceeded, etc. Both "NotDone" and "Unknown" will require some exception handling at the application level (Pm or Tm) and an accompanying reason can provide substantial help for this.

The above semantics guarantees that an action or transaction was either done or not done at all, but does not guarantee that the invoker will be informed of the outcome of the action or transaction. The responsibility to find out the outcome of an action or transaction rests with the invoker rather than with the underlying system. However, the underlying system guarantees that the action or transaction was not done more than once. If action lists and transaction lists are kept in stable storage, the guarantee holds even in the presence of Dm and Tm failures.

6.3.5 Stable Storage

Stable storage survives failures of storage devices and physical nodes. It is required for recovering from Dm and Tm failures. Every operation on stable storage must be atomic, so that data stored on stable storage can never be corrupted. This is equivalent to the **save** operation of [Gray 81, Lampson 81, Liskov 83, Loques 84]. Stable storage can be implemented as redundant logs on disk or tape [Gray 78], as duplexed disk pages [Lampson 81, Sturgis 80], or as memory units with backup power supplies [Spector 83]. Stable storage can exist on another physical node and be used via remote communication [Moss 81, Liskov 83]. It can also be obtained by means of active and passive redundancy which associates a module with an identical backup module on another physical node [Loques 84]. Active and passive redundancy was chosen for the implementation of the various levels of reliability of the model for two reasons: first, no recovery actions are necessary, the only delay involved is the time to detect a failure; and second, no special devices

are needed (e.g. disks) and this is advantageous in the target environment which consists of simple microcomputers.

According to [Loques 84], stable storage is obtained by associating a module with an identical **backup** module that runs on another physical node. The backup does not perform any application or database processing but is fully prepared to undertake such a role in case the active module fails. The backup provides two main functions. It is used to save state information from the active module (i.e. checkpoints for backward recovery) and can provide recovery from failure by detecting the failure of the active module and then by undertaking its role. The active module will have to save its state on its backup, be able to initialise a new backup, and periodically send "I-am-alive" messages to the backup so that the backup does not take over. Failure detection is done by a suitable timeout [Liskov 83, Loques 84]. If the active module does not save its state or send an "I-am-alive" message within the timeout period, the backup takes over. A new backup will then have to be created somewhere else in the system. Some of the system management components and associated algorithms that can be used to reconfigure the system in the presence of module failures can be found in [Loques 84, Kim 84]. Note that active and passive redundancy has often been used for providing recovery from physical node failures [Borr 81, Gray 81, Kopetz 82], but not for providing stable storage.

6.3.6 Dm Failures

To achieve transparent recovery from Dm failures, each Dm is associated with a backup Dm' on another physical node. The Dm' is used to support the various levels of reliability allowed by the model. The Dm' provides stable storage and backward recovery from failure. The difference between the various levels of reliability lies in the frequency of accesses to stable storage. Cold reliability never uses stable storage. Warm reliability accesses stable storage periodically in order to save the values of the data objects. Hot reliability accesses stable storage on a per action or transaction basis in order to save the values of data objects, the action list, and some state information whenever a transaction is prepared to commit as part of the 2PC.

The solution described in the rest of this section refers to the most general and stringent case to be implemented by a Dm, namely hot

reliability together with strong consistency. All other cases are simple subsets of this general solution. For efficiency reasons, access to stable storage (Dm') may be deferred until prepare or commit time. An extra prepare action must be provided by a Dm in order to support the 2PC that is used by a Tm to coordinate the commitment of a transaction across a number of Dm's. The actions that a Dm may provide to its clients (Pm's and Tm's) are:

* Get Address: If it refers to a base or copy data object, it returns a pointer to the data object. This is not available on a Dm which implements strong consistency.

* Immediate Read: Read the data object if it is not already locked by another transaction. No need to access stable storage or to record this action in the action list.

* Immediate Write: Carry out the write action if the data object is not already locked by another transaction. Access stable storage to save the data object and this action in the action list.

* Immediate Transaction: Such a transaction is implemented by the Dm and accesses only data objects contained within the Dm. It is treated similarly to immediate write actions if it contains at least one write action, else it is treated similarly to immediate read actions.

* Lock: Reserve the lock if the data object is available. No need to access stable storage or to record this lock action in the action list. The mode of locking (read or write) can be supplied as a parameter to a lock action.

* Deferred Read: Read the data object if it is already locked by the transaction. No need to access stable storage or to remember this read action in the action list.

* Deferred Write: Carry out the write action tentatively on the current version of the data object, if the data object has already been locked by the transaction. Record this write action in the action list. Defer access to stable storage until prepare time.

* Deferred Transaction: It is treated similarly to deferred write actions if it it contains at least one write action. Otherwise, it is treated similarly to deferred read actions.

* Prepare: Stable storage is accessed to carry out the write actions in the form of **batched writes** [Spector 83, Liskov 83, Allchin 83], to save this prepare action in the action list, and to save some state information (Tid and locks). Read locks are released at prepare time.

* Commit: This requires access to stable storage in order to commit the batched writes of the transaction, and to save this commit action in the action list. Write locks are then released.

* Abort: If the abort action is invoked during the first phase of the 2PC, stable storage need not be accessed at all. Otherwise, stable storage will have to be accessed to abort the transaction. All locks are released at abort time.

Immediate read, write and transaction are single message transactions which avoid the overheads of separate lock, prepare and commit actions for efficiency reasons. Immediate read and write actions are particularly efficient for accessing constructed data objects. Lock, prepare, commit and abort actions are required for supporting medium and strong consistency even in the presence of Dm failures.

It can be seen that the following save actions are needed for accessing stable storage (Dm'): Commit Writes Immediately, Prepare Batched Writes, Commit Batched Writes, and Abort Batched Writes [Allchin 83]. All the state information of a Dm (Tid's, locks, current and shadow versions) is held in volatile storage until the transaction prepares to commit. If a Dm fails, the state information of all transactions in the first phase of the 2PC will be lost and these transactions will have to be aborted. This is similar to the approach taken in [Liskov 83, Allchin 83].

6.3.7 Two Phase Commit Protocol (2PC)

The 2PC is required for supporting medium and strong consistency in the presence of Dm failures. It is used by a Tm to coordinate the commitment of a transaction across a number of Dm's. The Tm which is responsible for the execution of the transaction takes the role of the coordinator and the Dm's involved take the role of the participants. During the first phase, the Tm invokes lock, read and write actions and perhaps performs some application processing. If all is fine at the end of the first phase, the Tm requests from each Dm to prepare to commit the transaction. If a Dm can commit the transaction, it saves the results of the transaction on stable storage (Dm') and replies that it is ready to

commit. If all the Dm's reply that they are ready to commit, the Tm can proceed to commit the transaction. Otherwise, the Tm must abort the transaction. This is the end of the first phase. During the second phase, the Tm invokes commit or abort actions on the Dm's, depending on the outcome of the first phase. Before a Dm returns its reply, it accesses stable storage (Dm') and commits the results of the transaction. Receiving a reply from all the Dm's denotes the end of the 2PC as well as the end of the transaction. A more detailed description of the 2PC can be found in [Kohler 81, Sturgis 80, Lindsay 79, Gray 78].

The proposed solution supports decentralised synchronisation and recovery in the sense that each Dm is responsible for the data objects which it contains, and that each Tm is responsible for its transaction. Dm's support only local consistency, whereas the 2PC allows Tm's to ensure global consistency among a number of Dm's [Moss 81]. The 2PC can cope with Dm failures. In conjunction with Tm backups, it can ensure that a transaction is eventually committed or aborted everywhere.

6.3.8 Tm Failures

To achieve transparent recovery from Tm failures, each Tm is associated with a backup Tm' on another physical node. The backup Tm' provides stable storage and recovery from failure. This is required for supporting medium and strong consistency in the presence of Tm failures. Each Tm is assumed to be a well-formed transaction and to use two phase locking correctly. If a Dm uses lock timeouts, it may unilaterally abort a transaction, so a Tm will have have to use the 2PC to commit its transaction. The same 2PC is also used to commit a transaction in the presence of Dm falures. Transaction abort by a Dm and Dm failure during the first phase of the 2PC are treated similarly as transaction aborts.

A Tm must save its state on stable storage (Tm') at three points during its execution. First, when the transaction is invoked, its TransactionId and input parameters must be saved. Neither Tid nor ActionId's need be saved since they can be reconstructed by the Tm'. The second save occurs at the end of the first phase of the 2PC. If all the Dm's involved in the transaction reply that they are ready to commit, the state of the transaction is marked as "prepared" and is saved on the Tm' together with the results of read actions and any computations done by the Tm. Otherwise, the state of the transaction is marked as "aborted" and is

saved on the Tm'. The third and last save is at the end of the second phase of the 2PC when a transaction is committed or aborted everywhere.

If a Tm fails during the first phase of the 2PC, the Tm' will take over, mark the transaction as "aborted", and invoke abort actions on all the Dm's. Then, the Tm' may reply to the invoker that the transaction was aborted, or alternatively retry the transaction after incrementing the TransId of section 6.3.2 (i.e. retry the transaction as a new transaction). If a Tm fails during the second phase of the 2PC and the transaction was marked as "committed", the Tm' will take over and invoke commit actions (with the same Tid and ActionId's as the Tm) on all the Dm's. If a Dm has already committed the transaction, it will treat the request as a duplicate and return the reply from its action list. If a Dm has not already committed the transaction, it will first commit the transaction and then return its reply. Thus, the transaction is eventually committed and the saved results are returned to the invoker. If a Tm fails during the second phase of the 2PC and the transaction is marked as "aborted", the Tm' will take over and invoke abort actions (again with the same Tid and ActionId's as the Tm) on all the Dm's. If a Dm has already aborted the transaction, a successful reply is returned as abort actions are idempotent. If a Dm has not already aborted the transaction, it will abort the transaction and return its reply. Thus, the transaction is eventually aborted. If lock timeouts are implemented and the Tm fails or the transaction is aborted during the first phase of the 2PC, no abort actions need be invoked by the Tm' on Dm's, since lock timeouts will eventually abort the transaction everywhere.

6.3.9 Atomic Transactions

It is interesting to note that the mechanisms presented in the previous sections automatically provide for atomic transactions as have been advocated in the literature. A **transaction** is an application-oriented operation that is executed either in its entirety or not at all. In this sense, it is atomic. It provides a convenient unit for procedural abstraction, semantic integrity, concurrency control, and recovery from failures. A transaction is normally required to have the following properties [Spector 83, Allchin 83, Liskov 83 & 82, Gray 81]:

* Totality (Execution Atomicity): If a transaction cannot proceed during its execution, it must be able to abort and its partial results be undone.

* Durability (Permanence of Effect): The results of successfully completed transactions should never be lost, despite failures of storage devices and failures of physical nodes.

* Indivisibility (Concurrency Atomicity): If several transactions execute concurrently, they should not interfere with each other. Their effect should be as if they were executed in some serial order. This property is also known as serialisability.

* Resiliency (Failure Atomicity): Either all or none of the results of a transaction are done in cases of communication failures and physical node failures.

* No Cascading Aborts: This is required only for efficiency reasons. An incomplete transaction should not reveal its results to other transactions in order to avoid the overheads of cascading aborts if the transaction must be subsequently undone.

The recovery mechanism of concurrency control is adequate for achieving totality. Stable storage can provide durability. The synchronisation and recovery mechanisms of concurrency control can ensure indivisibility. Resiliency can be achieved by the mechanisms for recovering from communication, Dm and Tm failures together with the 2PC. Cascading aborts are avoided by holding locks until the end of a transaction.

Note that both concurrency control and Dm failures require a recovery mechanism. For this reason, an integrated solution has sometimes been proposed for both the recovery from concurrency control conflicts and the recovery from failures [Moss 81]. The solution proposed in this thesis, however, treats the recovery from concurrency control conflicts separately from the recovery from failures, similarly to [Allchin 83].

Nested transactions are allowed by the model. Since nested transactions are very expensive to implement in their full generality, and since most transactions in a realtime system are short, it was decided to implement the nested transactions of the model as single level transactions. The specification of a nested transaction according to the model may be transformed to an equivalent single level transaction by means of a preprocessor or even a compiler that provides inline code substitution.

A more detailed discussion on nested transactions and on ways for implementing them can be found in [Moss 81, Allchin 83, Loques 84].

6.3.10 Large Data Objects

It may happen that the parameters of an action or transaction cannot fit in a single message that is transmitted via the underlying communication system. In that case, a request or reply will have to be fragmented and transmitted as a series of messages. An appropriate transfer protocol will therefore be required. This protocol will have to use sequence numbers relative to the current action or transaction invocation. An efficient and fully operational protocol of this kind can be found in [Birrell 84]. The implementation of such a protocol can be in terms of communication filters between any two modules of Fig. 6.1. The same protocol is also required for the remote transfers of large data objects which are exported periodically or on change. If the implementation uses a heterogeneous environment, additional filters will be needed for data translation between physical nodes with different target machines. Such filters may be required only for remote actions and transactions.

6.4 The Conic Environment

Conic provides an integrated environment for the implementation and management of distributed systems [Kramer 83 & 85, Sloman 85]. It was chosen for the implementation of the realtime database model for three reasons: first, it is especially suited for developing distributed realtime systems; second it distinguishes between the programming of individual modules and the configuration of a system from instances of such modules; and third, it provides an easy-to-use and flexible base for experimentation. Conic comprises a hardware architecture, a distributed operating and communication system, a programming language, a configuration language, and a management system.

The programming language **Conic/P** [Kramer 84] is based on Pascal [Jensen 78] to which message passing has been added. The unit of programming and separate compilation is the **task module.** This corresponds to a Pascal sequential program. Modules have message passing interfaces declared as typed entryports and exitports. Arrays of ports are allowed. Two communication primitives are provided: request-reply and notify. The **request-reply** primitive allows bidirectional and synchronous message

passing. The **notify** primitive does not have a reply part and provides unidirectional and asynchronous message passing which does not block the sender. Messages are sent to local exitports and are received from local entryports. The request-reply primitive allows many exitports to be connected to the same entryport (many-to-one message passing), whereas the notify primitive allows an exitport to be linked to many entryports (one-to-many message passing). Messages from different exitports are queued at their destination entryport until they are received. There is also a **select** statement for selecting a single message from a set a entryports. A timeout can be used to withdraw from a select.

A separate configuration language **Conic/C** [Dulay 84] is used to build a system by creating instances of task modules and interconnecting exitports and entryports. Conic/C identifies the module types from which the system will be built, declares the instances of the types which will exist in the system, and describes the interconnection of the instances between their exitports and entryports. A module type used in a configuration specification may be a task module or a collection of modules called a **group module.** Thus Conic/C allows systems and subsystems to be built as hierarchical compositions of task and group modules in a nested manner. Like the task module, the group module may have a message passing interface consisting of typed entryports and exitports which can be linked to the ports of constituent module instances. Arrays of ports and arrays of module instances are allowed.

6.5 Prototype Implementation

Conic was used to implemenent the patient monitoring system of chapter 5. The task modules of Conic correspond to Pm's but were also used as the basis for implementing Im's, Tm's and Dm's. The transformation from the notation of the realtime database model to the Conic programming and configuration languages was manual, but the long term aim would be to provide automatic transformation by means of a computer-aided tool. All read, write, lock, commit and abort actions and transactions were mapped onto Conic request-reply message passing. Periodic and onchange actions were mapped onto Conic notify message passing. Database modules were mapped onto Conic group modules. Views could not be supported, so they were completely ignored for both data and database modules. Thus, all interconnections had to be carried out in detail in terms of ports.

The entire patient monitoring system was implemented as specified in chapter 5. The prototype was only a simulation and could accomodate up to four patients. It involved writing the code not only for Dm's and Tm's but also for Im's and Pm's. A number of Pm's had to be programmed to perform the simulation of the various sensors, to collect patient histories, and to provide a man-machine interface. The man-machine interface allowed the data objects related to any patient to be displayed on a terminal and to be modified selectively. Since Conic has not yet been available on a distributed configuration, the prototype had to be implemented on a single machine. As a result, remote actions and reliability could not be implemented. As a further consequence, retransmissions, action lists, transfers of large data objects, and data translation between different machines were not necessary. Dynamic reconfiguration was not implemented because it was not yet supported by Conic. Future version of Conic will support dynamic reconfiguration and this will be used to provide dynamic reconfiguration of the database.

Despite the above restrictions, a substantial part of the model was implemented. Simple and structured data objects of all kinds were supported: base, derived, copy and constructed. Data objects were implemented only in main memory. Initialisation of data objects, value constraints, and integrity constraints were all implemented. Pointer access via the get-address action was provided.

Both immediate and deferred read and write actions were implemented as well as lock, commit, abort, periodic and onchange actions. Immediate and deferred transactions were also implemented within a Dm. Since, reliability could not be implemented, Dm's only supported transactions for concurrency control. For simplicity, lock timeouts were not implemented, so the 2PC was unnecessary. A Tm implemented a single level transaction by means of two phase locking. There was no distinction between read and write locks. The get-address action together with locks and current and shadow versions of data objects were adequate to implement all the levels of consistency provided by the model.

No state information (data values, locks, Tid's, etc) was maintained for derived or constructed data objects, but was created dynamically when such a data object was accessed. A return code was supplied as an output parameter of every action and transaction. Locking together with the single primary update strategy of the model and the asynchronous

updating of secondary copies combines the advantages of centralised and distributed locking and can yield good response times [Cheng 84].

6.6 Performance Issues

In order to evaluate the performance of the prototype implementation with regard to storage space and response times, some measurements were taken for the Dm that implements the 'Pulse' data module specified in section 5.2. The code and data space requirements on PDP-11/44 or VAX are 2-3 Kbytes if no transactions are supported, and 7-8 Kbytes if transactions are supported for concurrency control.

6.6.1 Local & Remote Actions

The absolute response times (i.e. no multiprogramming) on PDP-11/44 for accesses to **local** data objects are 1.2-2.0 msecs for actions performed via message passing, and 10-20 μsecs for pointer access. These measurements represent average response times for 20,000 consecutive action invocations, include loop overhead, and are commensurate to those reported in [Magee 84, LeBlanc 83]. Most of the time seems to be spent in the underlying system switching messages between tasks. If Pm's access Dm's via Im's, it takes twice as long for local accesses. Based on the measurements reported in [Magee 84, LeBlanc 83, Nelson 81], the response times for accesses to **remote** data objects are expected to be 20-30 msecs or more, depending on the size of the data object. Microcode implementation of crucial parts of the underlying system and data modules may speed up response times substantially, up to an order of magnitude or even more [Spector 82, Nelson 81]. For example, with microcode implementation, local actions may take 0.1-0.2 msecs and remote actions may take 0.5-1.0 msecs.

6.6.2 Cost of Consistency

Weak consistency by means of pointer access is efficient to implement for both storage space and response times. Medium and strong consistency involve a substantial overhead in storage space and, if transactions are excluded, the impact on response times is minimal. The only exception is when medium and strong consistency is used together with hot reliability, in which case the response times will be delayed substantially because of the accesses to stable storage.

6.6.3 Cost of Reliability

The cost of reliability is high in storage space and most importantly in respose times. Storage space increases slightly for cold reliability, more for warm reliability, and substantially for hot reliability. Note that two copies of a module must be maintained, with each copy requiring extra code for failure detection, batched writes actions, etc.

Software implementation of cold and warm reliability is not expected to affect response times, although in case of failure a response may be delayed substantially. Software implementation of hot reliability seems to be very expensive. It may slow down responses by an order of magnitude and, in case of failure, much more than an order of magnitude. Microcode implementation and use of special hardware (e.g. dedicated communication lines and interlocks) can yield acceptable response times even for hot reliability, for example, in the region of 1-10 msecs. Implementation of hot reliability is slow because of the need to access stable storage. This slows down both the actions that access stable storage (e.g. immediate write) and the actions that do not (e.g. lock or immediate read) but are still queued at the Dm interface. Concurrent invocation of actions on a Dm may reduce these unnecessary delays.

6.6.4 Cost of Atomic Transactions

The main problem with the implementation of a Tm across multiple remote Dm's (providing for concurrency control, communication and physical node failures) is that the code for both Tm's and Dm's is much more than the normal (no concurrency control, no failures) case and response times deteriorate significantly. The execution of a Tm may take a long and almost unpredictable amount of time. In addition, it may not be possible any more to achieve fast response times for a Dm since a transaction can never be aborted during the second phase of the 2PC.

Atomic transactions require both strong consistency and hot reliability, thus they are quite expensive to implement [Liskov 83, Allchin 83, Moss 81] both in storage space and response times. As a result, the actual use of atomic transactions is still questionable for distributed realtime systems. More research is required in the use and validity of atomic transactions in distributed realtime systems.

Given that the plant state changes so quickly, it is pointless to use atomic transactions for obtaining consistent snapshots of such data. It seems appropriate though to use transactions for atomic writes on a group of data objects, especially for those updated by human users.

Constructed data objects are certainly useful not only for allowing overlapping data objects and for modelling display frames, but also for providing fast access to a number of data objects via immediate read and immediate write actions. This provides a simple and efficient means for accessing remote data (for example, all the data objects of a remote controller) in a single message transaction. For normal accesses, such requests are well within the bound of the messages transmitted via existing communication systems. For similar reasons, transactions that are confined and implemented within a single Dm, are both useful for providing application-oriented operations and efficient to implement.

6.7 Chapter Summary

This chapter presented an architecture of a few basic components that can be interconnected to provide the overall functionality of a data module. Read and write actions were cast in the form of synchronous request-reply message passing or (remote) procedure calls. Periodic and onchange actions were cast in the form of asynchronous and unidirectional message passing. A number of run-time mechanisms were described for supporting the various performance characteristics provided by the model. These mechanisms include pointer access for fast response times, concurrency control, recovery from communication failures, module backups together with a two phase commit protocol for recovering from physical node failures, provision of atomic transactions and transfers of large data objects. Some of these mechanisms and all the functionality of a data module were implemented and tested in a prototype implementation of the patient monitoring system that was specified in chapter 5. A number of issues related to the implementation cost of these mechanisms were also discussed.

CHAPTER 7

DISCUSSION

This chapter discusses a variety of issues related to the realtime database model and to the design process that led to its derivation. It starts by highlighting the influences and rationale behind the design of the model. Then, it proceeds to compare the data module with an abstract data type and to discuss possible extensions and uses for the model. A realtime system development method that embodies the model is also presented together with some criteria that can be used for the decomposition of data objects and their allocation into data modules.

7.1 Design Influences & Rationale

The traditional definition of a database model is that it comprises data structures, operations on the data structures, and integrity constraints that supplement the data structures and operations [Manola 83]. It has also been suggested that a database model should encompass the definition of views [Hammer 79] and provide facilities for application-oriented operations [Brodie 81, Smith 80]. The concept of a data module follows on these trends, but caters also for the performance characteristics and the distributed nature of realtime systems. More specifically, the data module was designed to provide a unit that would accomodate as many of the following issues as possible:

(1) modular specification.
(2) logical grouping of several data objects.
(3) encapsulation of data, operations, views, constraints, performance.
(4) physical distribution of data and operations.
(5) consistency with regard to integrity constraints and transactions.
(6) replication and recovery for reliability purposes.
(7) dynamic reconfiguration
(8) reusable specification (and software) in many different contexts.
(9) identical interface for local and remote operations.
(10) incremental or even separate compilation.

Thus, the data module can be used to specify a variety of concepts: data objects, views, constraints, transactions and performance. Alternatively one could opt for separate data modules, view modules and transaction modules in order to specify data, views and transactions respectively. The latter seems more practical for implementation purposes. However, the model retained the single data module concept for simplicity reasons and, after all, separate specifications for data, views and transactions can be derived easily from the proposed specification of a data module.

The model allows sharing of data types at specification time and sharing of data objects at run time. For data types, the concept of a system-wide definitions pool was introduced. This pool was not structured in any way because it was felt that this function falls in the domain of the configuration database. The data object concept corresponds to the record concept in existing database models, i.e. the unit of access and manipulation of data. The model allows for user-defined data objects and these data objects can be of variable granularity, for example, from a single sensor reading to an entire log. An additional, hidden benefit for implementation purposes is this: by associating a lock with a data object, one can automatically obtain locks at different granularities.

The view concept of the model is novel. Existing database models tend to consider a view as a structural filter on one or more data objects, yielding a single data object. The proposed view concept is different. A view is a collection of operations on data objects. This novel view concept was introduced to provide interface control between modules. Note that the concept of exported and imported views is similar to the export and import statements of various programming languages, and also similar to the export and import schemas of the federated database approach described in [Heimbigner 85].

The model may be seen as providing abstraction from the underlying hardware components [Schell 71, Seitz 85]: processing modules correspond to processing elements (e.g. microprocessors, transputers, controllers); data modules correspond to storage devices (e.g. main memory, disk, bubble); and interactions between modules correspond to communication media (e.g. buses, serial links, local area networks). This brings the hardware level distinction between processing elements and storage devices at the level of the software architecture.

The data module was chosen as the unit of replication for reliability purposes. The usual hardware solution to the problem of reliability is to associate a physical node or controller with a duplicate backup. In case of failure, the backup will take over. The reliability mechanism adopted for a data module is a straightforward copying of this hardware solution [Loques 84]. However, the unit for efficiency (response time) copies of data was chosen to be a data object because efficiency copies are required selectively at many different places. Replicating the entire contents of a data module would be prohibitively expensive.

At the configuration level, the model adopted a mixed approach combining the hierarchical composition of a database with the efficiency and flexibility of a network. Hierarchical composition is achieved via nested specifications of database modules. This provides a means to cope with complexity and reflects the hierarchical organisation of realtime systems. Nested views can be used to accomodate additional hierarchical views within the overall hierarchy of the database. The view concept is also useful within a database module. It allows network interconnections among the components of a database module and can be used to bypass the strict hierarchy of the database by means of lateral interconnections.

Database modules and views are conceptual entities for specification and mapping purposes. They are used only at (re)configuration time to carry out the various bindings, so they incur no run-time overheads. At run time the database will be just a network of data modules, thus all the run-time overheads will be due to data modules and communication delays. The model however forces neither a hierarchy nor a network on the design of a realtime database. It is up to the designer to choose which structure is most appropiate for his particular system.

Note that the model can handle both distributed and centralised databases equally well. A distributed database must be specified as a collection of data modules. The model provides the flexibility to allocate a data module on any physical node of a distributed realtime system. A centralised database can be specified in two ways: either as a single data module which is very efficient with regard to time and space, but also monolithic, so any change will require the entire database to be reconfigured; or as a collection of data modules, which is less efficient but much more amenable to partial reconfiguration.

The model has four important properties. First, the same concepts and notation are used to specify a database at various levels of abstraction or refinement. The concepts and notation are slightly different only when data modules are eventually reached at the lowest level. Second, a single unifying concept, the data module, can be used to specify a variety of concepts: data objects, views, constraints, transactions and performance. Third, the concept of a "nested" entity has been applied consistently wherever it was possible: nested definitions of data types and objects, nested transactions, nested views, nested database modules, and nested specifications of performance characteristics. Fourth, the model is upwards compatible with a configuration database since the specification of a module can be readily integrated into a semantic model, for example, a semantic network of nodes with each node describing a different part or aspect of the system.

7.2 Comparison with Abstract Data Types

The concept of an abstract data type (ADT) encapsulating data together with the operations which manipulate it, is well known [Liskov 74 & 77] and several researchers have already tried to apply it to database systems [Brodie 81, Baroody 81, Liskov 83, Claybrook 85]. In this way, application programming may be seen as an extension of database manipulation, and database manipulation as an extension of database definition. The similarity between a data module and an ADT is notable: the data section defines the data that is encapsulated within the ADT, the transactions give the operations to be performed on the ADT, and the constraints correspond to the ADT invariant. A data module however is more general and flexible than an ADT because it uses views to interface with other modules and includes performance considerations.

A data module, like an ADT, provides representation abstraction from the underlying storage devices and access methods, which is similar to that provided by the internal view of the ANSI architecture [ANSI 78]. For example, the client of a data module will be unaware that the data resides in main memory or on disk, or that indices or hash tables are used for accessing the data. Further abstraction in the form of ADT operations can be obtained by means of transactions on a data module.

The performance characteristics of a data module provide an additional, new kind of abstraction which may be called **performance abstraction.** It refers primarily to the domain of distributed realtime systems and applies only to well understood and well developed mechanisms. The implementation of these performance characteristics can be provided automatically when the actual code for the database management system is generated. At the moment only response time, reliability and consistency characteristics have been formalised in this way. As further mechanisms are developed, it may become possible to formalise additional performance characteristics such as availability targets and concurrency control mechanisms. Such performance characteristics may be considered as **options** that are available to the designer of a realtime database to obtain different performance tradeoffs for his particular system. The advantage of these options is that they are specified in a declarative manner, and that their implementation may eventually be automated.

A serious problem with ADTs as pointed out in [Smith 80] is that they suppress both representation details and structural information of the real world. For example, an ADT would suppress the fact that a unit contains three turbines, see Fig. 4.10(c). In addition, operations on constituent ADTs are routed invariably via the enclosing ADT. This leads to a vast number of operations for the ADTs at the higher levels. Such operations are usually tedious to specify and may incur unnecessary run-time overheads. Unlike the composition of ADTs from other ADTs, the model uses a separate configuration language to specify a database as a composition of data modules. In this way, the structure of the database is not embedded in the specification of the data modules but is specified separately in a declarative, not procedural manner, as advocated in [Magee 84, Nelson 81]. The view concept is of great help here by providing abstraction from the detailed data objects, actions and transactions that are encapsulated in the view. This allows data and database modules to be interconnected easily and thus leads to simple specifications at the configuration level.

7.3 Possible Extensions

Several extensions are possible to the realtime database model presented in chapter 4. These extensions have not been included in the model for simplicity reasons. The DDL can be extended in various ways in order to increase the functionality and flexibility of the data module

concept. Instantiation parameters can be used to initialise (part of) the data in a data module, and symbolic constants (e.g. MaxSize=10) can be introduced to increase the readability and consistency of the specifications. Both instantiation parameters and constants can be considered as meta data that can be used to define data objects, initial values, integrity constraints, or to be referenced within a transaction.

An additional "sequence" or "relation" construct could be used for modelling data structures with unlimited storage capabilities (e.g. some kinds of histories, logs and archives). The problem with such structures is that response times can never be guaranteed, so they may be misused for modelling plant state data with critical realtime constraints. One solution to this problem is to declare their maximum cardinality at specification time so that worst-case response times can be calculated. If sequences or relations are adopted, the processing modules and transactions will require additional facilities for manipulating this new kind of data structures in ways similar to those described in [Hoare 72, Schmidt 77 & 83, Reimer 83]. The data type relation also allows additional configuration data to be modelled.

Exports of data objects on change or periodically can be considered as a very limited form of triggers, i.e. spontaneously activated operations when a condition becomes true. More general facilities for triggers that can invoke complex operations (e.g. transactions) could be added to a data module. A data module will then turn into an "active monitor" with the ability to carry out automatic collection and maintenance of histories and logs, alarm and event processing, arithmetic calculations, etc. Although such an extension would add useful functionality to a data module, it was omitted because similar tasks can be performed by processing modules and because it would blur the fundamental distinction between processing and data modules as advocated in this thesis.

Security restrictions in the form of access control lists can also be added to a data module. A convenient description could be to associate each client of the database with one or more client classes, and then associate each client class with one or more of the views exported from a data module. Many other approaches are also possible. Security restrictions can be used for checking purposes at (re)configuration time before a binding (linking) is performed, or at run time whenever dynamic access to a data object is attempted.

Unlike the DDL, only a few extensions are envisaged for the DCL. Instantiation parameters, symbolic constants, and security restrictions in the form of access control lists can be added to a database module. These are similar to the extensions suggested for the DDL. The DCL can also be extended to specify the logical-to-physical mapping of database components by means of an **at node(Id)** construct [Dulay 84, Liskov 83], where "Id" is either the name of a physical node or the name of another module instance. In the latter case, the new module instance will be created at the physical node at which the "Id" module instance resides. This will allow data or database modules to be mapped into the nodes of a physical configuration at instantiation time.

7.4 Use of the Model

The proposed model has been applied successfully to four case studies. Two of these case studies concerned electricity generation plants and extracts from these specifications appeared in chapter 4. The third case study referred to an industrial process control system. The fourth and last case study was a patient monitoring system and the complete specification was given in chapter 5. A prototype was also developed for this patient monitoring system in order to investigate the practical implementation problems and to measure the realtime performance of such an implementation. The conclusions of the above case studies together with the performance measurements from the prototype implementation indicate the viability and practicality of the proposal.

The realtime database model can be used in several ways: it can serve as a formal design tool for specifying a realtime database; it can provide a precise documentation and communication medium for the system designers and the human users of the database; and it can also lay the foundation for a new kind of database management system, cf. [Wulf 80, Hilfinger 81, Hammer 81, Kerth 84]. In the long term, an interactive computer-aided tool (or set of tools) is envisaged to help with the use of the model. Such a tool may guide a designer step by step during the specification of a realtime database, analyse and check a specification for syntactic correctness and consistency, store and perhaps maintain versions of specifications, validate the functional and performance characteristics of a specification, and eventually generate code automatically from DDL, DCL and other specifications. This tool could also be used for entering and validating database modifications, and initiating the operations necessary for dynamic reconfiguration.

7.5 A RealTime System Development Method

The realtime database model is of rather limited use unless it can be incorporated within a system development method. Since processing and data are closely related in a realtime system, such a method must allow the design of the processing and data to proceed in a parallel and interdependent manner. Such methods however are still research topics [Dowson 84], therefore this section presents only a rough outline of a system development method that embodies the proposed model. The main steps of this tentative method are given below. More details on the subject can be found in the literature [Yao 85 & 82, Gomaa 84, Agosti 84, Ceri 84, Jackson 83, Teorey 82, Chen 80, Lum 79]. This method also serves as an introduction to the criteria for the decomposition and allocation of data, which are discussed in section 7.6.

* Requirements Specification: An abstract model of the environment based on the logical, physical or administrative organisation of the plant can be used as a framework in which to cast the functional and performance requirements of the system, cf. [Jackson 83]. This includes the control strategy to be employed and details about the man-machine interfaces. The output of this step is a data-flow diagram of the system, consisting of processing objects and flows of data objects between them, cf. [Gomaa 84]. Note also that specific functional and performance requirements are associated with each object.

* Conceptual Design: Top-down refinement based on functional or data-driven decomposition is applied to each of the processing and data objects. The process of decomposition terminates when elementary objects such as a procedure, simple array or record are eventually reached. The criterion for these successive decompositions is to reduce complexity, i.e. every object at any level of refinement should be easy to understand. The output of this step comprises all the elementary data and processing objects together with the relationships between them.

* Logical Design: During this step, the elementary processing and data objects are allocated into processing and data modules. Before doing so, it may be necessary to decompose these elementary objects further, for example, in order to facilitate subsequent reconfiguration or for performance reasons. The criteria that can be used for the further decomposition of data objects and for their subsequent allocation into

data modules are discussed in section 7.6. A module is considered to be the unit of concurrency in the system and will reside at a single physical node. Copy data objects and reliability backups are expected to be defined as part of the logical design. The output of this step is the specification of all the processing and data modules and their interconnection, i.e. a complete logical description of the system. Every module now comprises both the functions to perform and the performance required. The DDL and DCL can be used respectively for the specification and interconnection of the data modules.

* Physical Design: Each data module is now designed in detail with decision being made about storage devices, access methods, reliability schemes, and details about the physical configuration of the system. The output of this step is a complete physical description of the system, including the detailed design of each module (cf. "software blueprint", the system is now ready for implementation).

* Implementation: The system is implemented using existing computer technology, each processing or data module being realised by a sequential software task. This step includes the usual chores of programming, testing, integration, and verification of the functional and performance capabilities of the system. As it has been mentioned already, the long-term aim is to achieve automatic generation of the various software modules and of the overall system from DDL, DCL and other specifications by means of a computer-aided tool. This implies that there must be standard ways for generating software modules and for selecting a suitable system architecture.

Note that there is some commonality between the concepts of the proposed model and those found in two existing software development methods, DARTS and JSD. DARTS is specialised for the design of realtime systems [Gomaa 84], whereas JSD is claimed to be a general-purpose development method [Jackson 83]. In the DARTS context, processing modules correspond to "tasks", data modules to "data stores", read and write actions to "closely-coupled message passing", and the update actions to "loosely-coupled message passing". In the JSD context, processing modules correspond to "function processes", data modules achieve the effect of "state-vector connections" in a more general manner, and the read, write and update actions may be seen as "data-stream connections".

7.6 Criteria for Decomposition & Allocation

During the conceptual design of a realtime database, the criterion for the successive decompositions is to **reduce complexity**, i.e. every object at any level of refinement must be **easy to understand.** Put another way, the effort of understanding or decomposing an object should be approximately constant at each point of refinement, cf. notion of "constant entropy" in [Simon 62]. Each object should therefore be decomposed to a few constituent objects as it has been suggested in [Miller 56] and supported by the "SA Maxim" in [Ross 77]. The process of decomposition terminates when elementary objects are reached, an elementary object being easily understood on its own.

During the logical design, the elementary processing and data objects will have to be allocated into processing and data modules respectively. The **allocation criteria** that can be used for allocating data objects into data modules and for further decomposing data objects are:

* Locality of Reference: Data objects that are always used by the same processing module(s) should be placed in a single data module [Toth 78, Franta 81]. If disjoint parts of an object are used by many processing modules, it will be necessary to decompose this object further.

* Logical Relationship: Objects that refer to the same entity or contribute towards the same overall function, will better display this relationship if they are grouped together in a single data module. Objects that are not related should be placed in different data modules. This criterion is similar to "functional cohesion" used for allocating processing objects into processing modules [Gomaa 84, Stevens 74].

* Performance Characteristics: All the objects in a data module are supposed to exhibit comparable performance characteristics. Conversely, an object should be split into two or more objects if these objects are required to have different performance characteristics. Other factors such as module size may be treated similarly, for example, if a data module is too big to fit on a single physical node, it should be split into two or more data modules.

* Geographical Proximity: A data module will reside entirely at the same physical node. An object must be split into two or more objects if these objects are likely to reside at different physical nodes [Toth 78].

* Easy Change & Reconfiguration: The objects in a data module should be expected to undergo mutual change at comparable reconfiguration intervals. If parts of an object are to be reconfigured independently of each other, it will be better to decompose this object further.

* Reusability: A data module should be self-contained so that it can be reused in a variety of contexts without any change [Wegner 83]. An object should be split into two or more objects if these objects are likely to be reused independently of each other.

* Processing Modules: There must be some structural correspondance between the configuration of the data modules and the configuration of the processing modules in a system (cf. [Jackson 75]). Since data modules reflect the shared data between processing modules, such correspondance need not be one to one. Normally there will be fewer data modules in a system than processing modules, but care should be taken to avoid making data modules "stress points" in a system, cf. [Beane 84]. For example, a system comprising a single data module and a dozen processing modules is rather unbalanced, with the single data module being an obvious stress point. If a system is unbalanced, some modules will have to be split further and others to be combined together.

It must be emphasised here that these allocation criteria also become decomposition criteria whenever an object is allocated into a data module. As a consequence, these criteria may have to be applied recursively until no more decomposition is necessary. In practice, a heuristic can be used when applying these criteria: an allocation can be considered as successful if four or more of the criteria are satisfied for the data module concerned. These criteria should be taken more as design guidelines rather than as absolute determinants of a good design.

The criteria of logical relationship, performance characteristics, geographical proximity, easy change and reconfiguration, and reusability can also be used for allocating processing objects into processing modules. Some further criteria for allocating processing objects into processing modules can be found in [Gomaa 84]. For allocation purposes therefore, a transaction can be treated similarly to a data object. For example, a transaction that is likely to change independently of the data objects it manipulates, should be placed in a separate data module. Transactions that define an abstract data type should be placed in the

same data module together with the data objects they manipulate.

It is worth pointing out that the criteria of logical relationship, easy change and reconfiguration, and reusability are similar to the criteria used for abstract data types [Liskov 74] and information hiding [Parnas 72 & 84]. However, applying the criteria of reusability, easy change and reconfiguration is not easy because future uses and changes are hard to predict. "It requires that the designer estimates the likelihood of changes. Such estimates are based on past experience and may require knowledge of the application area as well as an understanding of software and hardware technology" [Parnas 84].

7.7 Chapter Summary

This chapter discussed a variety of issues related to the realtime database model and to the design process that led to its derivation. The data module unifies together a variety of concepts: data objects, actions and transactions on these objects, integrity constraints, imported and exported views, and performance considerations. The similarity between a data module and an abstract data type is notable, but a data module is more general and flexible. Unlike abstract data types, the model does not embed the structure of the database within data modules but a separate configuration language is used to specify a database as a composition of data modules in a declarative manner. This is greatly facilitated by the view concept that provides the interface between modules. A database can also be specified hierarchically as nested specifications of database modules. Nested views can be used to provide different hierarchical views of the database. At run-time the database is just a network of data modules because nested database modules and views are only conceptual entities with no run-time overheads. The model can be extended easily to accomodate additional concepts such as relations, instantiation parameters, symbolic constants, security restrictions, and logical-to-physical mapping. It has been intended primarily for use as a tool for the specification and automatic generation of distributed realtime databases. It can also be incorporated in a realtime system development method and be used in conjunction with some criteria for decomposing data objects and allocating them into data modules.

CHAPTER 8

CONCLUSIONS

8.1 Summary of Achievements

This thesis has applied database concepts to the domain of distributed realtime systems. It started with a survey of the general requirements for realtime systems and proceeded to identify the need and particular requirements for both a realtime database and a configuration database. Detailed examination of existing database models showed that these models do not satisfy all the requirements for a realtime database. As a result, a new and improved model was proposed.

The model separates the definition of data modules from the configuration of a database module as interconnected instances of data modules. Views are used to provide interface control between modules. A database module can be specified in terms of other database modules, thus allowing nesting of database specifications. The separation between data definition and database configuration provides the flexibility to carry out changes in the form of dynamic reconfiguration. A data module also provides performance abstraction, i.e. abstract treatment for response times, reliability backups, and levels of consistency. A component architecture, essentially a refinement of the model, can be used to implement the overall functionality of a data module.

The design and validation of the model was based on four case studies. One of these case studies was a patient monitoring system for which a prototype implementation was developed. The conclusions of the case studies, supported by experiences and performance measurements from the prototype implementation, indicate the viability and practicality of the approach. In addition, a semantic model was suggested as the best available candidate for a configuration database.

8.2 Critical Evaluation

The proposed realtime database model constitutes a synthesis of ideas and experiences from several computing disciplines: realtime systems, distributed systems and databases, application-oriented specifications, programming languages and software engineering. Its importance lies in the fact that it provides a unified collection of concepts and a notation for the logical specification of a realtime database.

The central theme of the model is the separation between data definition and database configuration. This separation facilitates the development of the database and provides the flexibility in adapting to changes. The database is decomposed to a set of data modules with well-defined interfaces between them. Individual data modules are easier to understand, specify, implement and modify. They can also be compiled separately. As a result, the entire database is easier to design, specify, implement and maintain. A configuration specification records the interactions between data modules explicitly so that modifications can be carried out in an orderly and safe manner. The approach is novel in that it can allow dynamic reconfiguration of the database without shutting down the whole database or system.

More specifically, the model meets the realtime database requirements of chapter 2 and does not have the limitations of the database models examined in chapter 3. A data module caters for both simple and structured data and can describe plant state, logs, histories, some configuration data, and display frames. Data types and specific data objects can be handled explicitly, and there are facilities for specifying initial values and derived data. Flexible access to data is supported as well as transactions for application-oriented operations. Access to data is normally by name and hierarchical naming is achieved by means of nested views and nested specifications of database modules. A number of views can be specified on a data module. A view restricts access to the data objects and operations offered by a data module and provides precise interface control between modules.

A variety of response times, reliability backups, levels of consistency, and integrity constraints can be specified. These provide different options to a designer in order to allow tradeoffs between response times, reliability, integrity and flexibility. Only backward recovery to a previously saved state can be provided automatically by the model. Forward recovery requires processing modules to explicitly update backup data modules with state information from the environment. Such forward recovery is obviously dependent on the particular environment. Support for forward recovery could be provided by allowing only write actions and rejecting read actions on a backup data module for a short period immediately after recovery to a saved state.

Extensions and modifications can be accomodated easily in the form of dynamic reconfiguration with minimal disruption. In fact, the data module forms a convenient unit for dynamic reconfiguration and distribution. A number of extensions to the model are possible: relations, initialisation parameters, symbolic constants, security restrictions and logical-to-physical mapping.

Admittedly, the model is not very good for configuration data in the sense that it does not yield the best and most elegant solutions. A data module can represent static collections of data objects quite well but this has problems with addition and removal of data objects. For example, the physical configuration of a system could be represented as an array of records, with each record containing the details of a physical node in the system. Transactions would then be required to delete, update or insert (up to the limits of the array) information about individual physical nodes. These transactions effectively simulate the operations provided by relational or semantic models. Of course, the model could be extended to provide facilities similar to those of relational and semantic models, but this was thought unnecessary since such facilities will have to be provided anyway by the configuration database of the system. Nevertheless, the realtime database model could be used to implement parts of the configuration database, if necessary.

8.3 Design Principles Revisited

It is also worth reviewing the design principles laid out in chapter 1, in the way they are fulfilled by the realtime database model.

* Consistency: The same basic concepts were used throughout the design of the model. The same concepts and notation can specify a database at different levels of abstraction. The concepts and notation are only slightly different when data modules are eventually reached. The concept of a nested entity was applied consistently to data types, data objects, views, transactions, database modules, and the specification of performance characteristics. The concepts of the model integrate nicely with each other, thus ensuring the unity and ease of use of the model.

* Simplicity: A simple dichotomy between data definition and database configuration was made. A module consists of seven sections, with each section comprising only a few concepts. For example, a single concept is

associated with imported data types, four kinds of data objects are possible (base, derived, copy and constructed), and three performance characteristics can be specified. The single data module concept can be used to specify a variety of data, transactions and views. The component architecture that implements the model uses only a few basic components.

* Utility: Both the consistency and simplicity of the model contribute to its ease of use and indicate that it is applicable to large and complex real world situations. This is supported by the conclusions of the case studies and the experience from the prototype implementation. The model exhibits a good tradeoff between simplicity and functionality: its concepts are both simple and useful for the majority of cases in the problem domain. A more complete realtime system development method will certainly give more guidance in the application of the model.

* Flexibility: Concepts can be easily removed or added to the model. Removal of a module section has minimal impact on the other sections, for example, removal of constraints or performance. Removal of a concept from a section has minor impact on the remaining concepts, for example, removal of initial values or constructed data objects. Additional concepts can be included either as a separate section or within an existing section. For example, security restrictions and symbolic constants can be added as new sections, whereas relations can be added to the data section of a data module, and logical-to-physical mapping to the instance section of a database module. In addition, the model can be incorporated in a realtime system development method, and is compatible with a configuration database in the form of a semantic model.

8.4 Suggestions for Further Work

The research work presented in this thesis can be continued and extended in several directions which are listed below according to priority:

(1) More experience with prototype implementations is essential for evaluation purposes. In particular, hot reliability and strong consistency must be implemented, aiming initially for functional correctness and then for various efficiency optimisations. Such an implementation will encompass read and write locks, lock timeouts, transfers of large data objects, retransmissions and action lists to overcome communication failures, and backup modules that use a two phase commit protocol to cope with physical node failures.

(2) The issue of dynamic reconfiguration needs further investigation and experience with running systems. Two areas that require more work are: first, the algorithms for analysing and validating a configuration change especially with regard to nested changes; and second, the atomicity of changes both on the configuration database and on the running system.

(3) The concepts and notation for the specification of processing modules need investigation as well as the integration of data and processing modules within a realtime system. The specification of the processing modules could use, for example, an event-based model of computation, augmented perhaps with petri-net concepts. Views should be incorporated within processing modules so that they can access data modules. The specification of processing modules should ultimately incorporate exported views, imported views, imported data types and performance characteristics. Thus, the specification of a realtime system will require a data definition language for data modules, a processing definition language for processing modules, and a single configuration language for instantiating and inter-connecting data and processing modules. The database configuration language of the proposed model can be used as the basis for the configuration language of the system.

(4) An interactive computer-aided tool can be developed for supporting the development and maintenance of realtime databases and realtime systems in general. Such a tool may guide the designer step by step during the specification of an entire system, analyse and check specifications for syntactic correctness and consistency, store and perhaps maintain versions of specifications, and generate code automatically from specifications. This tool could be used both for entering specifications and for carrying out dynamic reconfigurations. As a consequence, it will require access to other tools. One of these tools will be the configuration database of the system, which will maintain a pool of data types, a pool of data modules, a pool of processing modules, system descriptions, etc. The tool may also employ graphical representations for displaying, entering and editing the configuration specifications of a system.

(5) Such a tool can also be extended to analyse the functional and performance characterictics of a realtime database or system. For example, it may employ animation for checking the functionality and deadline calculations for checking the response times. There is obviously a need for additional algorithms that can analyse and verify reliability and consistency characteristics. Such algorithms will have to be robust and amenable to automation.

(6) Research into the abstract treatment of further performance characteristics is necessary in order to take full advantage of the performance abstraction provided by a data module. The performance characteristics which may be considered are concurrency control strategies (pessimistic or optimistic), other recovery from failure mechanisms, availability targets, safety considerations, quiescent state when dynamic reconfiguration may be carried out, etc.

(7) The concept of a configuration database is still in its infancy and will continue evolving for some time. A more detailed definition of its data characteristics may be necessary as well as collection of sample data for case studies in order to select and validate the most suitable database model. The final step, of cource, will be a prototype implementation for a configuration database.

(8) A more rigorous and refined realtime system development method will certainly be useful. Existing methods are not intended for distributed systems and focus almost entirely on functionality. The perspective method should be able to handle both functional and performance characteristics equally well. It should also include criteria for the decomposition of processing into functions and for the allocation of functions into processing modules (tasks). The derivation of such a method is likely to be evolutionary rather than revolutionary [Dowson 84].

(9) There is already a need for the integration of independently existing realtime databases or systems. Integration could proceed, firstly, by merging the respective configuration databases together and secondly, by joining the actual systems together. This can take the form of dynamic reconfiguration without shutting down anyone of the systems involved. The view concept can be of particular help here. For example, a new view could be defined for each existing system and then an aggregate view could be defined in terms of these views in order to provide access to all the existing systems. Many other solutions are possible. MAP [MAP 84] also aims at solving this problem of integration, but unfortunately is very much oriented towards implementation, its view of a realtime database is severely limited, and does not offer any high-level help with the software management aspect of distributed realtime systems.

8.5 Final Remarks

The main contribution of this thesis is a proposal for a new realtime database model. This model constitutes a synthesis of ideas and experiences from many computing disciplines and is novel in several ways. The definition of database components is separated from the configuration of the database as interconnected instances of components. This separation facilitates the development of the database and provides the flexibility to carry out changes in the form of dynamic reconfiguration without shutting down the whole database or system. Views are used to provide interface control between components and nested database configurations are allowed. The model also provides abstract treatment for the performance characteristics of the database. Some of the concepts of the model may be also applicable to other areas of computer science. For example, the distinction between data definition and database configuration may influence the integration of existing commercial databases and guide the development of new distributed databases. The view concept and the abstract treatment of performance characteristics may have repercussions on the design tools and the programming languages for distributed systems in general.

APPENDIX

FORMAL DEFINITIONS

The Data Definition Language (DDL) and the Database Configuration Language (DCL) are formally defined here in an extended form of traditional BNF. Syntactic constructs appear either as single words, for example, constant, or as multiple words linked together by hyphens, for example, data-object-declaration. The null sequence of symbols is denoted by empty. From the terminal symbols, keywords are underlined, for example, module, but all special characters appear unaltered, for example, ; , <-- , < , etc.

Optional constructs are enclosed in curly brackets and alternative constructs are separated by a vertical bar as shown below:

{ ; data-object-id } -- zero or more repetitions

array-type | record-type -- alternative selections

The term "definition" is used when a template is defined from which multiple instances can be created, for example, data-type-definition, whereas the term "declaration" is used to indicate instance creation, for example, data-object-declaration.

Note that some of the constructs appearing on the right-hand side of syntax rules are enclosed in exclamation marks. These constructs are not defined in this appendix either because their definition can be found in the Pascal Manual [Jensen 78], for example, !Pascal scalar type!, or because their detailed and rigorous definition is beyond the scope of this thesis, for example, !any first order predicate involving data objects together with some predefined functions such as old, new, etc!.

A1. BNF Syntax for the Data Definition Language (DDL)

```
data-module-definition ::= data module data-module-id ;
                               export-section
                               import-section
                               type-section
                               data-section
                               constraint-section
                               transaction-section
                               performance-section
                           end.

data-module-id  ::=  !Pascal identifier!


export-section  ::=  empty
                   | export  exported-view  {  ; exported-view  }  ;

exported-view   ::=  data-object-id    : action { , action }
                   | transaction-id    : transaction
                   | exported-view-id  : view-definition

view-definition ::=  view  view-part  {  ;  view-part  }  end

view-part ::=  data-object-id { , data-object-id } : action { , action }
             | transaction-id { , transaction-id } : transaction
             | imported-view-name { , imported-view-name } : view

action ::=  read
          | write
          | periodic update-period
          | onchange onchange-condition

update-period      ::=  empty  |  unsigned-number  time-units

time-units         ::=  msecs  |  secs  |  mins  |  hours

onchange-condition ::=  empty  |  first-order-predicate

imported-view-name ::=  imported-view-id  instance-specifier

instance-specifier ::=  empty  |  range-specifier  |  index-specifier

range-specifier ::= [ lowbound..upperbound { , lowbound..upperbound } ]

index-specifier ::= [ unsigned-integer { , unsigned-integer } ]

lowbound   ::= unsigned-integer

upperbound ::= unsigned-integer


import-section ::=  empty  |  import imported-view { ; imported-view } ;

imported-view  ::=  imported-view-id { , imported-view-id }
                    : array-part  imported-view-definition

array-part     ::=  empty  |  array  range-specifier  of
```

```
imported-view-definition ::=  module-type-id  .  exported-view-name

exported-view-name       ::=  exported-view-id
                              { . exported-view-id  instance-specifier }

type-section  ::=  empty  |  type  data-type-id  { ; data-type-id } ;

exported-view-id  ::=  !Pascal identifier!

imported-view-id  ::=  !Pascal identifier!

data-type-id      ::=  !Pascal identifier!

data-object-id    ::=  !Pascal identifier!

transaction-id    ::=  !Pascal identifier!

module-type-id    ::=  !Pascal identifier!


data-section ::=  empty  |  data  data-object-declaration
                            {  ,  data-object-declaration  }  ;

data-object-declaration  ::=  base-data-object
                            | derived-data-object
                            | copy-data-object
                            | constructed-data-object

base-data-object ::= data-object-id  {  , data-object-id  }
                     : data-type  values-constraint  initial-values

data-type  ::=   data-type-id  |  data-type-definition

data-type-definition ::=  simple-type  |  array-type  |  record-type

simple-type  ::=  integer | real | boolean | byte | character
                | string  ( unsigned-integer ) | enumerated-type

enumerated-type  ::=  !Pascal scalar type!

array-type   ::= array [ index-type { , index-type } ] of data-type

index-type   ::= !Pascal index type!

record-type  ::= record
                   fixed-part  { ; fixed-part }  { ; variant-part }
                 end

fixed-part   ::= record-field  { ; record-field }

record-field ::= field-id { , field-id } : data-type

field-id     ::= !Pascal identifier!

variant-part ::= empty  |  case-type

case-type    ::= case tag-field-id of
                   variant-field  { ; variant-field }  otherwise-field
                 end
```

```
tag-field-id ::= field-id

variant-field     ::=  case-label-list : ( field-definition )

case-label-list   ::= !Pascal case label list!

field-definition  ::= empty  |  field-id : data-type

otherwise-field   ::= empty  |  ;  otherwise : ( field-definition )

values-constraint ::= empty  |  ,  values  values-definition

values-definition ::= values-list  { , values-list }

values-list ::=  constant | constant .. constant | < values-definition >

constant    :: = !any Pascal constant including strings and bytes!

initial-values .    ::= empty  |  ,  initially  initialisation

initialisation      ::= constant  |  < initial-values-list >

initial-values-list ::= initialisation { , initialisation }

derived-data-object ::= data-object-id : simple-type  =  derivation

derivation  ::= !any Pascal expression involving constants and simple
                data objects  already  declared  in  the data module!

copy-data-object ::= data-object-id : data-type-id
                     <-- imported-view-id  imported-view-index
                     . data-object-id  copy-update-mechanism

imported-view-index ::= empty  |  index-specifier

copy-update-mechanism   :: = periodic  |  onchange

constructed-data-object ::=  data-object-id  :  data-type-id
                                                < component-list >

component-list ::=       component       { , component  }
                   |  < component-list  { , component-list } >

component ::=  data-object-id  |  constant

constraint-section  ::= empty  |  constraint   constraint-definition
                                              { ; constraint-definition } ;

constraint-definition ::= first-order-predicate

transaction-section ::= empty | transaction   transaction-definition
                                            { ; transaction-definition } ;

transaction-definition ::= transaction-id  formal-parameter-list ; block

block ::= !a Pascal-like block  allowing access to the data objects of
          the data module as global data via "assignment", "with" and
          "case"  statements,  and  also nested invocation  of remote
          actions  and  transactions,  or of other local transactions!
```

```
nested-invocation ::=
          imported-view-id imported-view-index . remote-invocation
        | local-invocation

remote-invocation ::=  action-invocation | transaction-invocation

local-invocation  ::=  transaction-invocation

action-invocation ::=  read   ( data-object-id ; variable )
                     | write  ( data-object-id ; variable )

transaction-invocation  ::=  transaction-id  actual-parameter-list

formal-parameter-list ::= empty | ( input-parameters output-parameters )

input-parameters      ::= in  parameter-list ;

output-parameters     ::= out parameter-list

parameter-list        ::= parameter-group { ; parameter-group }

parameter-group       ::= variable { , variable } : data-type

actual-parameter-list ::= empty | ( variable { , variable } )

variable              ::= !Pascal variable!

performance-section ::= empty | performance  response-time-definition
                                             reliability-definition
                                             consistency-definition

response-time-definition ::=  empty  |  response-time-clause ;

reliability-definition   ::=  empty  |  reliability-clause ;

consistency-definition   ::=  empty  |  consistency-clause ;

response-time-clause     ::=  response time : unsigned-number time-units

reliability-clause       ::=  reliability   : reliability-kind

consistency-clause       ::=  consistency   : consistency-kind

reliability-kind  ::=  none | cold backup | warm backup | hot backup

consistency-kind  ::=  weak | medium | strong

unsigned-number   ::=  unsigned-integer | unsigned-real

unsigned-integer  ::=  !Pascal unsigned integer!

unsigned-real     ::=  !Pascal unsigned real!

first-order-predicate  ::=  !any first order predicate involving
                            data objects together with some pre-
                            defined functions such as old, new, etc!
```

A2. BNF Syntax for the Database Configuration Language (DCL)

```
database-module-definition ::= database module database-module-id ;
                                   export-section
                                   import-section
                                   type-section
                                   instance-section
                                   constraint-section
                                   transaction-section
                                   performance-section
                               end.

database-module-id ::= !Pascal identifier!

export-section ::= empty | export exported-view { ; exported-view } ;

exported-view  ::= exported-view-id  :  view-definition

view-definition ::= view  view-part  { ; view-part }  end

view-part   ::= nested-view  { , nested-view }  : view

nested-view ::=   imported-view-name
                | module-instance-name . exported-view-name

module-instance-name ::= module-instance-id  instance-specifier

exported-view-name   ::= !identical to the DDL exported-view-name!

module-type-id       ::= !Pascal identifier!

module-instance-id   ::= !Pascal identifier!

instance-specifier   ::= !identical to the DDL instance-specifier!

imported-view-name   ::= !identical to the DDL imported-view-name!

import-section       ::= !identical to the DDL import-section!

type-section   ::= empty | type module-type-id  { ; module-type-id }  ;

instance-section ::= empty | instance   module-instance-declaration
                                      { ; module-instance-declaration } ;

module-instance-declaration ::=  module-type-id
 | module-instance-id { , module-instance-id } : array-part module-type-id

array-part ::= !identical to the DDL array-part!

constraint-section ::= empty  | constraint   constraint-definition
                                          { ; constraint-definition } ;

constraint-definition ::= first-order-predicate
```

```
link-section ::= empty | link  link-definition  { ; link-definition } ;

link-definition  ::=   module-instance-name . imported-view-name  to
                       module-instance-name . exported-view-name
                     | module-instance-name . imported-view-name  to
                       imported-view-name


performance-section ::= !identical to the DDL performance-section!

first-order-predicate ::= !any first order predicate  involving  module
                            types and instances  together with some pre-
                            defined functions, e.g. old, new, count, etc!
```

A3. BNF Syntax for Database Modification

```
modify-module-definition ::= modify module modify-module-id ;
                                modification { ; modification }
                             end.

modify-module-id  ::=  !Pascal identifier!

modification  ::=  insertion | deletion | replacement


insertion ::=
  insert export       exported-view { ; exported-view }
| insert import       imported-view { ; imported-view }
| insert type         module-type-id { ; module-type-id }
| insert type         module-type-id  : module-type-construction
| insert instance     module-instance-declaration
                                { ; module-instance-declaration }
| insert constraint   constraint-definition { ; constraint-definition }
| insert link         link-definition { ; link-definition }
| insert performance  response-time-definition
                      reliability-definition
                      consistency-definition


exported-view ::= !identical to the DCL exported-view!
imported-view ::= !identical to the DCL imported-view!

module-type-construction ::= modify-module-id ( module-type-id )

module-instance-declaration ::=
                    !identical to the DCL module-instance-declaration!

constraint-definition ::= !identical to the DCL constraint-definition!
link-definition       ::= !identical to the DCL link-definition!

response-time-definition::=!identical to the DCL response-time-definition!
reliability-definition  ::=!identical to the DCL reliability-definition!
consistency-definition  ::=!identical to the DCL consistency-definition!


deletion ::=
  delete export       exported-view-id      { ; exported-view-id      }
| delete import       imported-view-name    { ; imported-view-name    }
| delete type         module-type-id        { ; module-type-id        }
| delete instance     module-instance-name  { ; module-instance-name  }
| delete constraint   constraint-definition { ; constraint-definition }
| delete link         link-definition       { ; link-definition       }
| delete performance  response-time-definition
                      reliability-definition
                      consistency-definition


exported-view-id     ::= !Pascal identifier!
module-type-id       ::= !Pascal identifier!

imported-view-name   ::= !identical to the DCL imported-view-name!
module-instance-name ::= !identical to the DCL module-instance-name!
```

```
replacement ::=

  replace export      exported-view-id  by  exported-view-replacement
                  { ; exported-view-id  by  exported-view-replacement }

| replace import      imported-view-name  by  imported-view-replacement
                  { ; imported-view-name  by  imported-view-replacement }

| replace type        module-type-id  by  module-type-replacement
                  { ; module-type-id  by  module-type-replacement }

| replace instance    module-instance-name by module-instance-declaration
                  { ; module-instance-name by module-instance-declaration }

| replace constraint constraint-definition  by  constraint-definition
                  { ; constraint-definition  by  constraint-definition }

| replace link        link-definition  by  link-definition
                  { ; link-definition  by  link-definition }

| replace performance  response-time-replacement
                       reliability-replacement
                       consistency-replacement


exported-view-replacement ::=  exported-view | view-definition

imported-view-replacement ::=  imported-view | imported-view-definition

module-type-replacement   ::=  module-type-id
                             | module-type-construction
                             | module-type-id : module-type-construction


view-definition ::= !identical to the DCL view-definition!

imported-view-definition ::=
                  !identical to the DCL imported-view-definition!


response-time-replacement ::= empty
                  | response-time-clause  by  response-time-clause ;

reliability-replacement  ::=  empty
                  | reliability-clause  by  reliability-clause ;

consistency-replacement  ::=  empty
                  | consistency-clause  by  consistency-clause ;


response-time-clause ::= !identical to the DCL response-time-clause!
reliability-clause   ::= !identical to the DCL reliability-clause!
consistency-clause   ::= !identical to the DCL consistency-clause!
```

REFERENCES

[Adiba 78]: M. Adiba and others, Issues in Distributed Database Management Systems: A Technical Overview, Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, Germany, September 1978, pp. 89-110.

[Agosti 84]: M. Agosti and R. G. Johnson, A Framework of Reference for Database Design, ACM SIGBDP Data Base, Vol. 15, No. 4, Summer 1984, pp. 3-9.

[Allchin 83]: J. E. Allchin, An Architecture for Reliable Decentralised Systems, PhD Thesis, Georgia Institute of Technology, Technical Report GIT-ICS-83/23, September 1983.

[Altaber 81]: J. Altaber and F. Beck, The Distributed Database for the CERN SPS System, Computers in Industry, Vol. 2, 1981, pp. 105-107.

[Andrews 83]: G. R. Andrews and F. B. Schneider, Concepts and Notations for Concurrent Programming, ACM Computing Surveys, Vol. 15, No. 1, March 1983, pp. 3-43.

[Andriopoulos 83]: X. Andriopoulos, On the Suitability of the Existing Database Models and DBMS Architectures for Industrial RealTime Databases, EWICS TC13 Working Paper, XA-WP-8302, October 1983.

[Andriopoulos 84]: X. Andriopoulos, Issues on Industrial RealTime Databases, EWICS TC13 Working Paper, XA-WP-8401, January 1984.

[Andriopoulos 85]: X. Andriopoulos and M. Sloman, A Database Model for Distributed RealTime Systems, Research Report, Imperial College, Dept of Computing, DOC 85/8, June 1985.

[ANSI 78]: D. Tsichritzis and A. Klug (ed), The ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Database Management Systems, Information Systems, Vol. 3, No. 3, 1978, pp. 173-191.

[ANSI 84a]: ANSC X3H2, (Draft Proposed) Network Database Language, Document X3H2-84-23, March 1984.

[ANSI 84b]: ANSC X3H2, (Draft Proposed) Relational Database Language, Document X3H2-84-24, March 1984.

[Baroody 81]: A. J. Baroody and D. J. DeWitt, An Object-Oriented Approach to Database Implementation, ACM Transactions on Database Systems, Vol. 6, No. 4, December 1981, pp. 576-601.

[Barson 82]: I. C. Barson and others, A Distributed Database for Power Station Control and Information Systems, Proc. 2nd British National Conference on Databases, July 1982, Bristol U.K., pp. 122-132.

[Beane 84]: J. Beane, N. Giddings and J. Silverman, Qualifying Software Designs, Proc. 7th Int. Conf. on Software Engineering, Orlando, Florida, March 1984, pp. 314-322.

[Bernstein 81]: P. A. Bernstein and N. Goodman, Concurrency Control in Distributed Database Systems, ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 185-221.

[Birrell 84]: A. D. Birrell and B. J. Nelson, Implementing Remote Procedure Calls, ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, pp. 39-59.

[Blickley 84]: G. J. Blickley, New Developments in Distributed Process Control Systems, Control Engineering, Vol. 31, No. 11, October 1984, pp. 102-106.

[Borr 81]: A. J. Borr, Transaction Monitoring in ENCOMPASS: Reliable Distributed Transaction Monitoring, Proc. 7th Int. Conf. on Very Large Databases, Cannes, France, September 1981, pp. 155-165.

[Bragger 83]: R. P. Bragger, A. Dudler, J. Rebsamen and C. A. Zehnder, Gambit: An Interactive Database Design Tool for Data Structures, Integrity Constraints and Transactions, ETH, Institut fur Informatik, Report 55, September 1983, pp. 65-95.

[Bransby 82]: M. L. Bransby, Direct Digital Control in CEGB Power Stations, in Computer Control of Industrial Processes, S. Bennet and D. A. Linkens (ed), Peter Peregrinus, 1982, pp. 155-169.

[Brodie 81]: M. L. Brodie, On Modelling Behavioural Semantics of Databases, Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, France, September 1981, pp. 32-42.

[Brooks 75]: F. P. Brooks Jr, The Mythical Man-Month, Addison-Wesley, 1975.

[Cattell 79]: R. G. Cattell, An Entity-Based Database Interface, Xerox Report CSD-79-9, Palo Alto, August 1979.

[Cattell 83]: R.G. Cattell, Design and Implementation of a Relationship-Entity-Datum Data Model, Xerox Report CSL-83-4, Palo Alto, May 1983.

[Ceri 84]: S. Ceri and G. Pelagatti, Distributed Databases: Principles and Systems, McGraw-Hill, 1984.

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

[Chen 80]: P. P. Chen (ed), Entity-Relationship Approach to Systems Analysis and Design, North-Holland, 1980.

[Cheng 84]: W. K. Cheng, S. McRae and J. Murow, A Distributed Database Using the Primary Copy Concurrency Control Concept, Proc. Trends and Applications - Making Database Work, Maryland, May 1984, pp. 207-214.

[Clarke 83]: L. A. Clarke, J. C. Wileden and A. L. Wolf, Precise Interface Control, University of Massachusetts at Amherst, Computer and Information Science Dept, COINS Technical Report 83-26, August 1983.

[Claybrook 85]: B. G. Claybrook and A. M. Claybrook, Defining Database Views as Data Abstractions, IEEE Transactions on Software Engineering, Vol. SE-11, No. 1, January 1985, pp. 3-14.

[Codd 70]: E. F. Codd, A Relational Model for Large Shared Data Banks, CACM, Vol. 13, No. 6, June 1970, pp. 377-387.

[Codd 82]: E. F. Codd, Relational Database: A Practical Foundation for Productivity, CACM, Vol. 25, No. 2, February 1982, pp. 109-117.

[Crichton 83a]: E. R. Crichton and M. G. Willey, Database Management in a Distributed Process Control System, Proc. 12th IFAC/IFIP Workshop on RealTime Programming, Hatfield U.K., March 1983, Pergamon Press, pp. 9-14.

[Crichton 83b]: E. R. Crichton, Mapping of the KENT Distributed System on a DBMS, EWICS TC13 Working Paper, ERC-WP-8301, October 1983.

[Crowley-Milling 74]: M. C. Crowley-Milling, The Control System for the SPS, European Organisation for Nuclear Research, CERN Report LAB II - CO/74-1, May 1974.

[Crowley-Milling 79]: M.C. Crowley-Milling, The Data Module: The Missing Link in High-Level Control Languages, Proc. 3rd Int. Conf. on Trends in Online Computer Control Systems, Sheffield U.K., March 1979, pp. 63-65.

[Deen 83]: S. M. Deen and P. Hammersley, Proc. 2nd Int. Conf. on Databases, Cambridge U.K., September 1983, Wiley-Heyden.

[Delatore 85]: J. P. Delatore and others, The 5ESS Switching System: Operational Software, AT&T Technical Journal, Vol. 64, No. 6, July & August 1985, pp. 1357-1384.

[DeRemer 76]: F. DeRemer and H. H. Kron, Programming-in-the-Large versus Programming-in-the-Small, IEEE Transactions on Software Engineering, Vol. SE-2, No. 2, June 1976, pp. 80-86.

[Dowson 84]: M. Dowson and others, Distributed Data Processing Design Methodology, Final Technical Report, Imperial Software Technology, London, May 1984.

[Dulay 84]: N. Dulay, J. Kramer, J. Magee, M. Sloman and K. Twidle, The Conic Configuration Language, Version 1.3, Research Report, Imperial College, Dept of Computing, DOC 84/20, November 1984.

[Eswaran 76]: K. Eswaran, J. Gray, R. Lorie and I. Traiger, On the Notions of Consistency and Predicate Locks in a Database System, CACM, Vol. 19, No. 11, November 1976, pp. 624-633.

[Fisher 83]: D. G. Fisher, Computer Control of Decentralised Process Systems, Computers and Chemical Engineering, Pergamon Press, Vol. 7, No. 4, 1983, pp. 395-421.

[Franta 81]: W. R. Franta, E. D. Jensen, R. Y. Kain and G. D. Marshall, RealTime Distributed Computer Systems, Advances in Computers, Vol. 20, Academic Press, 1981, pp. 39-82.

[Frost 83]: Frost & Sullivan, European Market for Distributed Control Systems, Frost & Sullivan Report No. E644/C, 1983.

[Frost 84]: Frost & Sullivan, Instrumentation and Control Equipment Market in the Power Industry, Frost & Sullivan Report No. A1271/X, 1984.

[Gallagher 84]: L. J. Gallagher and J. M. Draper, Guide on Data Models in the Selection and Use of Database Management Systems, NBS Special Publication 500-108, National Bureau of Standards, January 1984.

[Gifford 81]: D. K. Gifford, Information Storage in a Decentralised Computer System, PhD Thesis, Stanford University, as Xerox Report CSL-81-8, Palo Alto, June 1981, Revised March 1982.

[Gomaa 84]: H. Gomaa, A Software Design Method for RealTime Systems, CACM, Vol. 27, No. 9, September 1984, pp. 938-949.

[Gray 76]: J. N. Gray, R. A. Lorie, G. R. Putzolu and I. L. Traiger, Granularity of Locks and Degrees of Consistency in a Shared Database, in Modelling in Database Management Systems, G. M. Nijssen (ed), North-Holland, 1976, pp. 365-394.

[Gray 78]: J. N. Gray, Notes on Database Operating Systems, in Operating Systems: An Advanced Cource, Lecture Notes in Computer Science, No. 60, Springer-Verlag, 1978, pp. 393-481.

[Gray 81]: J. Gray, The Transaction Concept: Virtues and Limitations, Proc. 8th Int. Conf. on Very Large Data Bases, Cannes, France, September 1981, pp. 144-154.

[Gueth 83]: R. Gueth and Th. Lalive d'Epinay, The Distributed Data Flow Aspect of Industrial Computer Systems, Proc. 5th IFAC Workshop on Distributed Computer Control Systems, May 1983, Pergamon Press, pp. 1-8.

[Gueth 85a]: R. Gueth, J. Kriz and S. Zueger, Broadcasting Source-Addressed Messages, Proc. 5th Int. Conf. on Distributed Computing Systems, Denver, Colorado, May 1985, pp. 108-115.

[Gueth 85b]: R. Gueth, J. Kriz and S. Zueger, Broadcast Protocols in Distributed Computer Control Systems, Proc. 6th IFAC Workshop on Distributed Computer Control Systems, Monterey, California, May 1985.

[Haase 81]: V. H. Haase, RealTime Behaviour of Programs, IEEE Transactions on Software Engineering, Vol. SE-7, No. 5, September 1981, pp. 494-501.

[Halling 81]: H. Halling, Recent Advances and Future Trends in Distributed Control Systems, Proc. IFAC/IFIP RealTime Programming Workshop, Kyoto, Japan, September 1981, Pergamon Press, pp. 1-13.

[Hammer 79]: M. Hammer and D. McLeod, On Database Management System Architecture, Technical Memo, MIT/LCS/TM-141, October 1979.

[Hammer 81]: M. Hammer and D. McLeod, Database Description with SDM: A Semantic Data Model, ACM Transactions on Database Systems, Vol. 6, No. 3, September 1981, pp. 351-386.

[Hansen 78]: P. B. Hansen, Distributed Processes: A Concurrent Programming Concept, CACM, Vol. 21, No. 11, November 1978, pp. 934-941.

[Heimbigner 85]: D. Heimbigner and D. McLeod, A Federated Architecture for Information Management, ACM Transactions on Office Information Systems, Vol. 3, No. 3, July 1985, pp. 253-278.

[Heninger 80]: K. Heninger, Specifying Software Requirements for Complex Systems: New Techniques and Their Application, IEEE Transactions on Software Engineering, Vol. SE-6, No. 1, January 1980, pp. 2-13.

[Hilfinger 81]: P. N. Hilfinger, Abstraction Mechanisms and Language Design, PhD thesis, Carnegie-Mellon University, Technical Report CMU-CS-81-147, June 1981.

[Hoare 72]: C. A. R. Hoare, Notes on Data Structuring, in Structured Programming by O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, Academic Press, 1972, pp. 83-174.

[Hooper 85]: J. W. Hooper, J. T. Ellis and T. A. Johnson, Distributed Software Prototyping with ADS, Proc. 8th Int. Conf. on Software Engineering, Imperial College, London, August 1985, pp. 216-223.

[Houser 83]: K. D. Houser, Data Highway Provides Database Management, Computer Design, Vol. 22, No. 13, November 1983, pp. 118-124.

[Israel 78]: J. E. Israel, J. G. Mitchell and H. E. Sturgis, Separating Data from Function in a Distributed File System, Xerox Report CSL-78-5, Palo Alto, September 1978.

[Jackson 75]: M. A. Jackson, Principles of Program Design, Academic Press, 1975.

[Jackson 83]: M. A. Jackson, System Development, Prentice-Hall, 1983.

[Jensen 78]: K. Jensen and N. Wirth, PASCAL User Manual and Report, 2nd edition, Springer Verlag, 1978.

[Jervis 77]: P. Jervis, Communication and Software Elements for Distributed Control Systems, Proc. IEE Int. Conf. on Distributed Computer Control Systems, Birmingham U.K., September 1977, pp. 97-103.

[Jervis 82]: P. Jervis, Distributed Databases in Process Control Systems, Working Paper, Contribution to EWICS TC8 Meeting, November 1982.

[Jervis 83]: P. Jervis, Comparison of ANSC X3H2 Proposed Network Standard with CUTLASS Database, EWICS TC13 Working Paper, PJ-WP-8301, November 1983.

[Jervis 84]: P. Jervis, Standards for Industrial RealTime Databases, Proc. EWICS, April 1984, pp. 97-113.

[Juhasz 83]: G. Juhasz, K. Kovacs and I. Sari, PCDB- A Process Control Database Management System, Proc. 12th IFAC/IFIP Workshop on RealTime Programming, Hatfield U.K., March 1983, Pergamon Press, pp. 27-30.

[Kent 78]: W. Kent, Data and Reality, North-Holland, 1978.

[Kerth 84]: N. K. Kerth, Software Tools Automate Structured Analysis, Electronics Week, Vol. 57, No. 19, August 1984, pp. 69-72.

[Khan 83]: B. Khan and A. Lewis, IRTB - An Alternative to RealTime Programming, Proc. 12th IFAC/IFIP Workshop on RealTime Programming, Hatfield U.K., March 1983, Pergamon Press, pp. 51-56.

[Kim 84]: W. Kim, Highly Available Systems for Database Applications, ACM Computing Surveys, Vol. 16, No. 1, March 1984, pp. 71-98.

[Kohler 81]: W. H. Kohler, A Survey of Techniques for Synchronisation and Recovery in Decentralised Computer Systems, ACM Computing Surveys, Vol. 13, No. 2, June 1981, pp. 149-183.

[Koller 81]: H. Koller and K. Fruhauf, A Database Management System for Industrial Process Control, Computers in Industry, Vol. 2, 1981, pp. 171-177.

[Kompass 84]: E. J. Kompass, Inside Honeywell's New TDC 3000 System, Control Engineering, Vol. 31, No. 5, May 1984, pp. 101-103.

[Kopetz 82]: H. Kopetz, F. Lohnert, W. Merker and G. Pauthner, The Architecture of MARS (MAintainable Realtime System), Report MA 82/2, Technical University of Berlin, April 1982.

[Kramer 79]: J. Kramer, The Design of Distributed Processing Systems Using Stable Modules, PhD Thesis, Imperial College, Dept of Computing, December 1979.

[Kramer 83]: J. Kramer, J. Magee, M. Sloman and A. Lister, Conic: An Integrated Approach to Distributed Computer Control Systems, Proc. IEE, Vol. 130, Part E, No. 1, January 1983, pp. 1-10.

[Kramer 84]: J. Kramer, J. Magee, M. Sloman, K. Twidle and N. Dulay, The Conic Programming Language, Version 2.4, Research Report, Imperial College, Dept of Computing, DOC 84/19, October 1984.

[Kramer 85]: J. Kramer and J. Magee, Dynamic Configuration for Distributed Systems, IEEE Transactions on Software Engineering, Vol. SE-11, No. 4, April 1985, pp. 424-436.

[Lampson 81]: B. W. Lampson, Atomic Transactions (Chapter 11, pp. 246-265) and Remote Procedure Calls (Section 14.9, pp. 365-370), in Distributed Systems: Architecture and Implementation, Lecture Notes in Computer Science, No. 105, Springer-Verlag, 1981.

[LeBlanc 83]: T. J. LeBlanc and R. P. Cook, An Analysis of Language Models for High-Performance Communication in Local Area Networks, ACM SIGPLAN Notices, Vol. 18, No. 6, June 1983, pp. 65-72.

[LeLann 83]: G. LeLann, On RealTime Distributed Computing, Proc. IFIP Congress, Paris, September 1983, North-Holland, 1983, pp. 741-753.

[Leveson 84]: N. Leveson, Software Safety in Computer-Controlled Systems, IEEE Computer, Vol. 17, No. 2, February 1984, pp. 48-55.

[Lindsay 79]: B. G. Lindsay and others, Notes on Distributed Databases, IBM Research Report RJ2571, San Jose, 14-7-1979.

[Liskov 74]: B. Liskov and S. Zilles, Programming with Abstract Data Types, ACM SIGPLAN Notices, Vol. 9, No. 4, April 1974, pp. 50-59.

[Liskov 77]: B. Liskov, A. Snyder, R. Atkinson and C. Schaffert, Abstraction Mechanisms in CLU, CACM, Vol. 20, No. 8, August 1977, pp. 564-576.

[Liskov 82]: B. Liskov, On Linguistic Support for Distributed Programs, IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982, pp. 203-210.

[Liskov 83]: B. Liskov and R. Scheifler, Guardians and Actions: Linguistic Support for Robust Distributed Programs, ACM Transactions on Programming Languages and Systems, Vol. 5, No. 3, July 1983, pp. 381-404.

[Litwin 79]: W. Litwin, Distributed Databases: A Way of Thinking About, INRIA Research Report MOD-I-027, May 1979.

[Loques 84]: O. G. Loques Filho, A Fault-Tolerant Distributed Computer Control System, PhD Thesis, Imperial College, Dept of Computing, February 1894.

[Ludewig 85]: J. Ludewig and others, SPADES - A Specification and Design System and its Graphical Interface, Proc. 8th Int. Conf. on Software Engineering, Imperial College, London, August 1985, pp. 83-89.

[Lum 79]: V. Lum and others, 1978 New Orleans Database Design Workshop Report, IBM Research Report RJ2554, San Jose, 13/7/79.

[Magee 84]: J. N. Magee, Provision of Flexibity in Distributed Systems, PhD Thesis, Imperial College, Dept of Computing, April 1984.

[Manola 83]: F. Manola and A. Pirotte, An Approach to Multi-Model Database Systems, in [Deen 83], pp. 53-75.

[MAP 84]: Manufacturing Automation Protocol (MAP) Specification, Revision 1, April 1984, General Motors Corporation.

[Marsh 79]: B. E. Marsh, An Engineer's Approach to Database Definition, Proc. 3rd Int. Conf. on Trends in Online Computer Control Systems, Sheffield U.K., 1979, pp. 165-169.

[Marti 83]: R. W. Marti, Integrating Database and Program Descriptions Using an ER Data Dictionary, in Entity-Relationship Approach to Software Engineering, C.G. Davies and others (ed), North-Holland, 1983, pp. 377-392.

[Mesarovic 70]: M. D. Mesarovic, Multilevel Systems and Concepts in Process Control, Proc. IEEE, Vol. 58, No. 1, January 1970, pp. 111-125.

[Miller 56]: G. A. Miller, The Magical Number Seven Plus or Minus Two: Some Limits on Our Capacity for Processing Information, Psychological Review, Vol. 63, No. 2, March 1956, pp. 81-96.

[Mitchell 79]: J. G. Mitchell, W. Maybury and R. Sweet, Mesa Language Manual, Version 5.0, Xerox Report CSL-79-3, Palo Alto, April 1979.

[Mitchell 83]: R. J. Mitchell and H. M. Robinson, Normalisation in Another Context, Proc. 12th IFAC/IFIP Workshop on RealTime Programming, Hatfield U.K., March 1983, Pergamon Press, pp. 21-26.

[Moss 81]: J. E. B. Moss, Nested Transactions: An Approach to Reliable Distributed Computing, PhD Thesis, MIT/LCS/TR-260, April 1981.

[NBS 82]: Computer Corporation of America (CCA), An Architecture for Database Management Standards, NBS Special Publication 500-86, National Bureau of Standards, January 1982.

[Nelson 81]: B. J. Nelson, Remote Procedure Call, PhD Thesis, Carnegie-Mellon University, as Xerox Report CSL-81-9, Palo Alto, May 1981.

[Oppen 81]: D. C. Oppen and Y. K. Dalal, The Clearinghouse: A Decentralised Agent for Locating Named Objects in a Distributed Environment, Xerox Report OPD-T8103, Palo Alto, October 1981.

[Parnas 72]: D. L. Parnas, On the Criteria to be Used for Decomposing Systems into Modules, CACM, Vol. 15, No. 12, December 1972, pp. 1053-1058.

[Parnas 84]: D. L. Parnas, P. C. Clements and D. M. Weiss, The Modular Structure of Complex Systems, Proc. 7th Int. Conf. on Software Engineering, Orlando, Florida, March 1984, pp. 408-417.

[Paxton 80]: W. H. Paxton, A Client-Based Transaction System to Maintain Data Integrity, Xerox Report CSL-80-3, Palo Alto, March 1980.

[Pearce 84]: V. Pearce, Distributed Control Comes of Age, Control and Instrumentation, Vol. 16, No. 6, June 1984, pp. 75-81.

[Penedo 85]: M. H. Penedo and E. D. Stuckle, PMDB - A Project Master Database for Software Engineering Environments, Proc. 8th Int. Conf. on Software Engineering, Imperial College, London, August 1985, pp. 150-157.

[Pirotte 77]: A. Pirotte, The Entity-Property-Association Model: An Information-Oriented Database Model, Technical Report R343, M.B.L.E. Research Laboratory, Brussels, Belgium, March 1977.

[Pygott 83]: C. H. Pygott, CAMEO: An In-Store Relational Database Machine, in [Deen 83], pp. 37-52.

[Reimer 83]: M. Reimer, Implementation of the Database Programming Language Modula/R on the Personal Computer Lilith, ETH, Institut fur Informatik, Report 55, September 1983, pp. 49-64.

[Ridjanovic 83]: D. Ridjanovic and M. L. Brodie, Action and Transaction Skeletons: High-Level Constructs for Database Transactions, ACM SIGPLAN Notices, Vol. 18, No. 6, June 1983, pp. 94-99.

[Ross 77]: D. T. Ross, Structured Analysis (SA) : A Language for Communicating Ideas, IEEE Transactions on Software Engineering, Vol. SE-3, No. 1, January 1977, pp. 16-34.

[Sandoz 82]: D. J. Sandoz, A Survey of Computer Control, in Computer Control of Industrial Processes, S. Bennet and D. A. Linkens (ed), Peter Peregrinus, 1982, pp. 1-17.

[Schell 71]: R. R. Schell, Dynamic Reconfiguration in a Modular Computer System, PhD Thesis, MIT/MAC/TR-86, June 1971.

[Schmidt 77]: J. W. Schmidt, Some High-Level Language Constructs for Data of Type Relation, ACM Transactions on Database Systems, Vol. 2, No. 3, September 1977, pp. 247-261.

[Schmidt 83]: J. W. Schmidt and M. Mall, Abstraction Mechanisms for Database Programming, ACM SIGPLAN Notices, Vol. 18, No. 6, June 1983, pp. 83-93.

[Schoeffler 84]: J. Schoeffler, Distributed Computer Systems for Industrial Process Control, IEEE Computer, Vol. 17, No. 2, February 1984, pp. 11-18.

[Seitz 85]: C. L. Seitz, The Cosmic Cube, CACM, Vol. 28, No. 1, January 1985, pp. 22-33.

[Senko 75]: M. E. Senko, Information Systems: Records, Relations, Sets, Entities and Things, Information Systems, Vol. 1, No. 1, 1975, pp. 3-14.

[Simon 62]: H. A. Simon, The Architecture of Complexity, Proc. American Philosophical Society, Vol. 106, No. 6, December 1962, pp. 467-482.

[Sloman 81]: M. Sloman, J. Magee and J. Kramer, System Management Facilities in Conic, Working Paper, Imperial College, Dept of Computing, September 1981.

[Sloman 82]: M. Sloman, J. Magee, J. Kramer and K. Twidle, Network Management Facilities in Conic, Research Report, Imperial College, Dept of Computing, DOC 82/14, September 1982.

[Sloman 83]: M. Sloman and X. Andriopoulos, Databases for RealTime Applications (a requirements survey), Research Report, Imperial College, Dept of Computing, DOC 83/9, March 1983.

[Sloman 84]: M. Sloman, J. Magee and J. Kramer, Building Flexible Distributed Computing Systems in Conic, Proc. SERC Conf. on Distributed Computing Systems, September 1984, Peter Peregrinus, pp. 86-106.

[Sloman 85]: M. Sloman, J. Kramer and J. Magee, The Conic Toolkit for Building Distributed Systems, 6th IFAC Workshop on Distributed Computer Control Systems, Monterey, California, May 1985.

[Smith 80]: J. M. Smith and D. C. P. Smith, A Database Approach to Software Specification, in Software Development Tools, W. E. Riddle and R. E. Fairley (ed), Springer-Verlag, 1980, pp. 176-204.

[Smith 82]: J. M. Smith and D. C. P. Smith, Principles of Database Conceptual Design, in [Yao 82], pp. 114-146.

[Spector 82]: A. Z. Spector, Performing Remote Operations Efficiently on a Local Computer Network, CACM, Vol. 25, No. 4, April 1982, pp. 246-260.

[Spector 83]: A. Z. Spector and P. M. Schwarz, Transactions: A Construct for Reliable Distributed Computing, Carnegie-Mellon University, Technical Report CMU-CS-82-143, January 1983.

[Stevens 74]: W. P. Stevens, G. J. Myers and L. L. Constantine, Structured Design, IBM Systems Journal, Vol. 13, No. 2, 1974, pp. 115-139.

[van der Stok 83]: P. D. V. van der Stok, Mapping of SPS Control System to a Relational Database, EWICS TC13 Working Paper, PVDS-WP-8301, November 1983.

[van der Stok 84a]: P. D. V. van der Stok, Extension of Modula/R to Fit Process Control Requirements, EWICS TC13 Working Paper, PVDS-WP-8401, February 1984.

[van der Stok 84b]: P. D. V. van der Stok, The Feasibility of a Relational Database Programming Language for Process Control, Proc. 1984 Realtime Systems Symposium, Austin, Texas, December 1984.

[Sturgis 80]: H. Sturgis, J. Mitchell and J. Israel, Issues in the Design and Use of a Distributed File System, ACM Operating Systems Review, Vol. 14, No. 3, July 1980, pp. 55-69.

[Taylor 76]: R. W. Taylor and R. L. Frank, CODASYL Database Management Systems, ACM Computing Surveys, Vol. 8, No. 1, March 1976, pp. 67-104.

[Teichroew 80]: D. Teichroew and others, Application of the Entity-Relationship Approach to Information Processing Systems Modelling, in [Chen 80], pp. 15-38.

[Teorey 82]: T. J. Teorey and J. P. Fry, Design of Database Structures, Prentice-Hall, 1982.

[Tillman 82]: P. R. Tillman, ADDAM - The ASWE Distributed Database Management System, in Distributed Databases, H. J. Schneider (ed), North-Holland 1982, pp. 185-196.

[Toth 78]: K. C. Toth and others, The ADD System: An Architecture for Distributed Databases, Proc. 4th Int. Conf. on Very Large Data Bases, Berlin, Germany, September 1978, pp. 462-471.

[Traiger 83]: I. L. Traiger, Trends in Systems Aspects of Database Management, in [Deen 83], pp. 1-21.

[Tripkovic 83]: S. Tripkovic, Database Relational Model in the Control of Dynamic Systems, Proc. 1983 RealTime Systems Symposium, Arlington, Virginia, December 1983, pp. 210-219.

[Tsichritzis 76]: D. C. Tsichritzis and F. M. Lochovsky, Hierarchical Data Base Management, ACM Computing Surveys, Vol. 8, No. 1, March 1976, pp. 105-124.

[Weber 83]: H. Weber, Object-Oriented Distributed Database Design, in [Deen 83], pp. 196-221.

[Wegner 83]: P. Wegner, Varieties of Reusability, Proc. Workshop on Reusability in Programming, Newport, RI, September 1983, pp. 30-44.

[Williams 78]: T. J. Williams, Hierarchical Control for Large Scale Systems: A Survey, Proc. 7th IFAC Congress, A Link between Science and Applications of Automatic Control, Helsinki, Finland, June 1978, Pergamon Press, Vol. 2, pp. 1393-1406.

[Wirth 77]: N. Wirth, Toward a Discipline of RealTime Programming, CACM, Vol. 20, No. 8, August 1977, pp. 577-583.

[Wulf 80]: W. A. Wulf, Trends in the Design and Implementation of Programming Languages, IEEE Computer, Vol. 13, No. 1, January 1980, pp. 14-25.

[Yao 82]: S. B. Yao and others (ed), Database Design Techniques I, Lecture Notes in Computer Science, No. 132, Springer-Verlag, 1982.

[Yao 85]: S. B. Yao (ed), Principles of Database Design, Volume I, Logical Organisations, Prentice-Hall, 1985.

Acknowledgements

Coming to the end of this thesis, I can now sit back and recollect all the people that suffered me during the last three years and helped me in a way or another to complete the research work described in this thesis.

First and foremost, I am deeply indebted to my supervisor Morris Sloman for his guidance, help and patience throughout the whole period of my research. Jeff Kramer and Jeff Magee gave me invaluable suggestions and constructive criticism for which I am grateful to both of them. Naranker Dulay and Kevin Twidle are certainly among the best Unix and Conic hackers I have seen: they have my admiration and thanks for their help with Conic while I was doing my prototype implementation. Special thanks must also go to Unit-C Ltd, Worthing, Sussex, U.K., for providing the research studentship that made this work possible.

Peter van der Stok of CERN provided important feedback and encouragement on an earlier draft of the realtime database model. Morris Sloman read multiple drafts of this thesis and made a number of useful suggestions. The comments and help of Ricardo Anido on issues of concurrency control and reliability are much appreciated. David Robinson thoroughly reviewed this thesis and suggested many improvements. In fact, he has contributed much more than this. He has been a good friend. I owe him much.

I would also like to thank Sebastian Danicic for his friendship and patience during our long and stimulating discussions; Eunice Edwards, Linden Rice, and Carol Wooldridge for their prompt and friendly service; Ellen Haigh for being an excellent librarian; and the whole Department of Computing at Imperial College for hosting me and putting up with me both as an undergraduate and as a postgraduate student.

Last but not least, I am grateful to my groovy music records and non-academic books, including many of Hermann Hesse, for keeping me sane and refreshed; to my parents and sister for their unfailing love and support; and to Verity for her patience and love.

over ...

Epilogue

Tiny Sparrows & Cosmic Angels

There are times
I am a tiny sparrow
quivering in the dark, cold space,
flying over the sea,
travelling over the land,
and
whispering that old elegy
of human
loneliness, agony and pain.

There are times
I am a cosmic angel
shining in absolute, divine power,
moving through an infinity of souls,
penetrating into a multitude of hearts,
and
singing that old tune
of human
love, hope and happiness.

And there are times
of real, intimate struggle,
when
I am neither a tiny sparrow
nor a cosmic angel,
when
loneliness, agony and pain
mingle with
love, hope and happiness.

So I wonder . . .
what should I call
times like these ?
Eternal Moments
or, perhaps,
Instantaneous Eternities . . .

X.A. 29-12-84