A Static, Transaction Based Design Methodology for Hard Real-Time Systems

BY

Philip M. Sleat

Department of Computer Science,
City University.

This thesis is submitted as part of the requirements for the degree of Doctor of Philosophy.

Dedication

This is dedicated to Laura, whose love and encouragement has steered me through the last two years; to our parents Doreen and Fred, Anne and Chris, for their continuous love, and support of our educations. Many, many thanks.

This is also dedicated to Holst, The Planets, Op. 32, for keeping me entertained whilst typing[1].

[1] I'm glad Earth wasn't included and that Pluto wasn't discovered in 1914 when the Planets were written... seven chapters were enough!

Contents

List of Figures

Acknowledgements

There are many people that I would like to thank for their help during my research. As a SERC 'CASE' student, I have come into contact with many professionals both in industry and academia, too many to thank individually. Some 'names' do however require acknowledgement and thanks. First of all at SEMA Group, I would like to thank Dr. Mike Christie, Mr. Brian Hardwick, Mr. Fraser Beady and Mr. Dave Britten for passing on their extensive knowledge of real-time systems design. After spending the last seven years at The City University, London for both my BSc. and research, there are many people to thank. Thanks must go to the department support staff; Nigel Mitchem for promptly answering all my stupid questions and Chris Marshall for converting my CASE tool to X-windows. My acknowledgements are extended to Majid Mirmehdi and Mohammad Nejad-Sattery for help with the content and layout of my thesis. Thanks go to Mr. Phil Winterbottom, now of AT&T Bell Labs and Professor Bernie Cohen for their comments and useful suggestions. Thanks must also go to Mr. Ken Jackson and Professor Mike Moulding for kindly reviewing the work and being so helpful with their comments. Finally, my greatest thanks go to my supervisor, Professor Peter Osmon, whose many ideas, enthusiasm and confidence in my abilities have made this possible.

Declaration

ABSTRACT

This thesis is concerned with the design and implementation stages of the development lifecycle of a class of systems known as *hard real-time systems.* Many of the existing methodologies are appropriate for meeting the *functional requirements* of this class of systems. However, it is proposed that these methodologies are not entirely appropriate for meeting the *non-functional requirement* of deadlines for work within these real-time systems. After discussing the concept of real-time systems and their characteristic requirements, this thesis proposes the use of a general transaction model of execution for the implementation of the system.

Whereas traditional methodologies consider the system from the flow of data or control in the system, we consider the system from the viewpoint of the role of each shared data entity. A control dependency is implied between otherwise independent processes that make use of a shared data entity; our viewpoint is known as the data dependency viewpoint. This implied control dependency between independent processes, necessary to preserve the consistency of the entity in the face of concurrent access, is ignored during the design stages of other methodologies. In considering the role of each data entity, it is possible to generate other viewpoints, such as the dataflow through the processes, automatically. This however, is not considered in the work.

This thesis describes a staged methodology for taking the requirements specification for a system and generating a design and implementation for that system. The methodology is intended to be more than a set of vague guidelines for implementation; a more rigid approach to the design and implementation stages is sought. The methodology begins by decomposing the system into more manageable units of processing. These units are known as *tasks* with a very low degree of coupling and high degree of cohesion. Following the system decomposition, the data dependency viewpoint is constructed; a descriptive notation and CASE tool support this viewpoint. From this viewpoint, implementation issues such as generating control flow; task and data allocation and hard real-time scheduling concerns, are addressed. A complete runtime environment to support the transaction model is described. This environment is hierarchical and can be adapted to many distributed implementations.

Finally, the stages of the methodology are applied to a large example, a Ship Control System. Starting with a specification of the requirements, the methodology is applied to generate a design and implementation of the system.

Chapter 1

Introduction

Mars, the Bringer of War

1.1 Overview

1.1.1 Real-Time Systems

This thesis is concerned with real-time systems. The term 'real-time system' has been defined by a number of authors in the literature [All81], [Ben88], [LA90], [BW89], [LM88], [Sta88]. In considering these, there is not one definition for a real-time system that sufficiently covers all members of this class of systems. Instead, the best that can be done in defining a real-time system is to describe those characteristics that stand it apart from non real-time systems.

According to [NS90], a real-time system interacts with its environment within certain timing constraints. The system has to respond to a set of conditions from some external environment within certain, predefined, response times. In addition, the system must carry out other tasks at regular intervals dictated by the system clock.

The existence of timing contraints, leads to a further division within the class of real-time systems. There are those real-time systems where the timing constraints are 'hard'; in a hard real-time system if any of the timing constraints are not met, the system has failed. There are also 'soft' real-time systems where it is desirable to satisfy as many of the timing constraints as possible but it is not catastrophic if some timing constraints are not met. The existence of timing contraints within a system does not necessarily make it a real-time system. Consider

the payroll system that must finish the payroll for a month before the next pay date of the employees. In this respect the payroll system could be considered real-time because there are temporal contraints on when work must be done. The definition of a real-time system must be refined somewhat. The timing constraints imposed are often difficult to achieve using the conventional hardware or software techniques of on-line systems. Often, 'clever-tricks' as dictated by system 'gurus' must be used to meet the deadlines.

Further characteristics of real-time systems include the need for high degrees of reliability. Often real-time systems work in safety critical environments where lives, or at best, large sums of money, are at stake should the real-time system fail in some way. If the system fails then greater costs than those to replace the systems are incurred. As an example of the increased reliability of a typical real-time system consider 'SIFT' a fault tolerent flight system [MSS82]. The requirements for this system state that the probability of life threatening failure must be no greater than 10^{-9} during a ten hour flight. This is equivalent to a *mean time between failures* of 10 million years assuming maintenance after each ten hour flight.

[HP88] states that the past and present events that a real-time system is subjected to, change its behaviour. This change may simply be altered output based on updated input. Alternatively, this change may require that some subset of the systems processes is now made redundant or that some subset is made active. For many real-time systems we have no way of knowing what system processes are going to be active at any time until the system is run.

This work is aimed at those systems that have 'hard' timing constraints. The system consists of a number of tasks that must complete before well specified deadlines. The system may also contain a number of 'soft' tasks which can be executed on a 'best effort' basis. The actions of each task are well specified before the system is defined. The actual tasks that are active at any particular moment in time not however specified prior to run-time. The set of active tasks is determined by past and present actions in the system. In addition, the real-time system may have the requirement for a high degree of reliability. Examples of this sort of real-time system occur in process control environments and C3 (Command, Control and Communications) systems for ships. Such systems are often termed 'embedded' real-time systems [Zav82]. These systems must often react to rapidly changing environments and have high degrees of reliability.

1.1.2 The System Life Cycle

The work presented in this thesis is concerned with one part of the 'life- cycle' of a real-time system. The life cycle of a computer system is often considered using some variation of the waterfall model. The stages of this model are as follows [Som89], [NS90]:

1. Requirements analysis and definition. The systems services, constraints and goals are established by consultation with the eventual users of the system.

2. Specification of requirements. A precise definition of the system services is stated. This statement is in a form which is understandable by the eventual system user (or procurer) as well as the systems engineers that are to build the system. The user can use the specification to check that the eventual system will meet the requirements. The systems engineers use the specification to aid the design and then prove that this design meets the requirements.

3. Design. This stage states the way in which the system services are to be provided. In addition, the environment in which the system is to operate is defined.

4. Implementation. The design is converted into actual software and hardware modules.

5. Testing. The implementation is tested to ensure correct operation as specified by the earlier requirements stage.

6. Commissioning. A further set of testing is carried out by the eventual users in the actual environment that the system is to execute in.

7. Maintenance. The system is changed as a result of the discovery of errors, ommisions and changes to the initial requirements.

Much work has been done to ensure that the requirements and specification documents are as complete and unambiguous as possible [Hoa85], [Hen80], [NS90], [Bat87], [WM86],

[Jac83]. Many of these approaches provide notations for expressing various aspects of the lifecycle of the system. They improve the communication paths between the system designer and eventual user and highlight problems and errors in a design early on. However, not many

of these methods concentrate on the design and implementation stages. The system designer often has to rely on his/her own experience, a set of heuristic guidelines and perhaps a set of notational tools, to generate the design.

1.2 Research Objectives

Existing methodologies consider the design of the real-time system from the traditional control or data flow viewpoints. This leads to a natural specification of the behaviour of the system. However, these viewpoints do not guide the designer in the quest to meet the *non-functional* requirements of the system. The main aim of the research is to consider the non-functional requirement of meeting hard real-time deadlines for systems that are well specified. The ability of existing methodologies to address this problem should be considered and the suitability of the data/control flow models of the systems discussed.

1.2.1 Scope of The Research

This thesis concentrates on the design and implementation stages of the system lifecycle. Given a precise and unambiguous specification of the requirements of the application, a staged methodology to lead to a design is proposed. This methodology, together with supporting notations, provides guidelines for the design of hard real-time systems and describes essential run-time structures for an implementation. The methodology is aimed at providing more than the set of heuristic guidelines for system partitioning as in other methodologies.

The methodology described in this work considers a real-time application from a different, but complementary, viewpoint to that taken by existing methods. Methodologies such as in [WM86], [Bat87], [?] consider the application from the flow of data viewpoint. Data flow diagrams are constructed to aid the designer decompose the application into more manageable units of processing. To implement these 'modules' the designer needs to consider the flow of control through, even finer grained, chunks of processing. The viewpoint taken by the methodology described in this work considers the system from the use each independent activity makes of a set of shared resources; this is known as the data dependency viewpoint and shows the relationship between independent activities through the use of shared resources. The flow of control through the components of each independent activ-

ity is automatically generated from the data dependency viewpoint.

One of the major problems with existing methodologies that consider the design and implementation stages of the system lifecycle, is that they concentrate on the *functional* requirements of the system, perhaps at the expense of the *non-functional* requirements. [Som89] describes the functional requirements of the system as those services which are expected by the user and the non-functional requirements as the constraints under which these services must be provided. For example, a functional requirement of a chemical control plant might be to shut down the plant if the temperature exceeds 100°C. An associated non-functional requirement might state that the plant must be shut down within 100 ms.

The work described in this thesis considers the temporal, non-functional, requirements of the real-time system as being very important from early on in the design process. The methodology uses the data dependency viewpoint to guide the design and the temporal requirements of the application to guide the implementation. Other non-functional requirements affect the temporal properties of the implementation. As an example, the provision of increased reliability through redundancy changes the system in that the redundant copies of software and hardware components need to be maintained. These non-functional requirements need to be considered carefully in the design and implementation stages of the lifecycle of the real-time system.

1.3 Plan of the Thesis

The following chapter provides further motivation for the study of real-time systems. An execution model, the transaction is presented and its suitability for use in real-time systems discussed. The chapter goes on to consider commercial database systems, both 'conventional' and real-time, that implement the transaction model and discusses their suitability for use in hard real-time systems.

Chapter 3 introduces a model for an executing real-time system. This chapter describes how the real-time system may be constructed from a set of independent 'tasks'. The chapter further describes how these tasks are constructed from sets of transactions that conform to the model presented in Chapter 2. The constraints under which these transactions may execute are discussed.

Chapter 4 is logically split into two sections. The first describes a step by step methodology for generating a design for an implementation of a

real-time system from the application requirements specification. The second part of the chapter describes two supporting notations; one to define the data dependencies between the otherwise independent tasks and the second to describe the control flow through the actions of a task that results from these data dependencies.

Chapter 5 describes the problem of ensuring real-time tasks meet their hard timing constraints. Some existing solutions to this problem, and their ability to meet the deadlines of all tasks are considered. In this chapter it is proposed that in order to meet successfully the deadlines of all tasks under all circumstances, some static analysis of the timing properties is required before the system is implemented. The chapter goes on to describe such a static analysis that uses information generated by the design methodology of Chapter 4. Following this the chapter describes, in overview, a hierarchy of scheduling mechanisms for executing the tasks and transactions in an implementation. This forms the basis of the run-time system for an implementation of the designs generated by the methodology of Chapter 4.

Chapter 6 compares aspects of the methodology with the equivalent in other, well established methodologies. The chapter goes on to consider other general requirements of real-time systems, such as high reliability, and discusses how the methodology and execution platform of chapters 4 and 5 meet up to these requirements. The chapter concludes with a discussion of the advantages and disadvantages of the new design methodology.

Chapter 7 summarises the work described in the preceeding chapters. In addition, the chapter describes areas of further research that are needed before a completely usable, and general, real-time design methodology is developed.

There are four appendices at the end of the thesis. The first appendix describes the use of a simple CASE tool developed to support the methodology. This tool is written in 'C' and runs under GEM and X-Windows. The CASE tool allows automatic generation of the notations that support the method and provides help for carrying out the static analysis necessary to evaluate the design of a real-time system. The second appendix describes a large, and relatively complex, real-time application : a ship control system. This application is used to demonstrate the steps of the methodology described in chapters 4 and 5 of the thesis. The third appendix describes some simple static examples that demonstrate the worst case analysis necessary to test the schedulability of a design. The final appendix describes the execution of the hierarchy of scheduling components that form the run-time

system for the eventual implementation of the real-time system.

Chapter 2

The Transaction Model in Real-Time Systems

Venus, the Bringer of Peace

2.1 Introduction

This chapter discusses the characteristics of the data that is stored in and used by real-time systems. It then introduces a model of computation that is embodied in most database management systems and has great potential for building reliable distributed computer systems. This computational model is the transaction. It is argued that a real-time system should be composed of these transactions. An 'off the shelf' database management system might then be usable as the data store for the real-time system. Although the transaction model is appropriate, it is not often feasible to use a conventional DBMS. The reasons for this are discussed. The non-deterministic parts of the transaction model, where it is difficult or impossible to predict accurately the influence of the model on the temporal properties of the system are then considered. Finally, it is proposed that, in order to build predictable real-time systems based on the transaction model, the design must consider, amongst other aspects, the concurrency control necessary between concurrent transactions.

2.2 Characteristics of Real-Time Data

2.2.1 A Historical Perspective

Today's large real-time systems have developed from traditional embedded systems. With this development has come a greater need for the control and manipulation of large volumes of real-time data. Early embedded systems typically used very small amounts of data. Memory was expensive and the early applications didn't demand the storage of a great deal of information. [Ast84] describes some of the early control systems. In 1959 a process control system controlled the flow of 26 chemicals, 72 temperatures and 3 pressures. The volume and complexity of the stored data in such systems was not great. The complexity of the applications has, however, changed over time. Today's real-time applications require the storage of large volumes of data and need to manipulate them in sophisticated ways. As an example, modern C3 systems [1] often maintain a track table [Tay89] that records the positions of any other vehicle within radar sight of the C3 system. A complete history of the positions is stored so that the 'track' or path of the other vessels may be monitored and future positions of the vessels predicted. Track tables in C3 systems can become very large indeed.

2.2.2 Real-Time Data

The data held by real-time systems has different properties from that held in conventional, non real time computer systems [Sta88], [Sle91]. Data in a real-time system typically has some subset of the following properties :-

- short lifetimes.
- often out of date.
- high update to read ratio.
- high availability.
- potential for large volume.
- predefined and limited query paths.

[1] C3 stands for Command, Control and Communications.

These properties are now explained. Data held in a real-time system often has a very short useful lifetime. This lifetime is the period in which the data accurately models, or reflects, the environment from which it originates. It is often the case that this external environment is changing very rapidly. The period during which a data entity in the real-time system correctly represents the environment is therefore short. As an example, consider the real-time system which is 'tracking' the position of some external vessel. If the vessel is an aircraft travelling at Mach 1 its position changes by approximately 300m every second. A data entity modelling this position becomes out of date very quickly. Compare this typically very short lifetime with that of a non real-time data entity such as a payroll balance. The external attribute i.e. monthly pay of an employee, changes very slowly. The data entity modelling the monthly pay has a lifetime of a month.

Associated with the short lifetime of real-time data is the characteristic that the data held by the system is very often out of date. If the environment being modelled changes faster than the computer system can read in and store the physical attributes of the environment then the computer system holds out of date data. At best, the computer system is a constant 'step' behind the physical environment. At worst, the computer system gets progressively more out of date with the physical environment. It is often a requirement that real-time applications must be able to tolerate some degree of 'staleness' of the data that is used.

The update to read ratio of real-time data is generally higher that that for non real-time data processing systems [Dix88b]. It is often the case that the real-time system spends most of its time keeping data entities as accurate as possible by continually updating them with fresh data from the environment. The data entities might be read only when the user wishes to prepare some report or inspect the state of some part of the controlled environment. In this case, the data entities are updated more often than they are read. Compare this with the non real-time payroll system where the data entities are updated once each month but inspected for the preparation of management reports every day. For this non real-time data, the update to read ratio is low.

Real-time data must be stored with an assured high degree of availability to the real-time applications. Long periods of 'downtime' during which the data is unavailable are typically not tolerated in real-time systems. As an example, consider a real-time air traffic control system. The data used by the application must be available, approaching 100% of the time. If some part of the system should fail then it is critical that the data be made available by alternate means as soon as possi-

ble. Lives are potentially at risk if the data is unavailable. Compare this with non real-time data. It is argued that no lives are lost if some payroll data is unavailable for a whole week out of a given four week period. In this situation, at worst, people are inconvenienced. [2]

Even though real-time systems typically model and control fairly restricted environments there is still the potential for the storage of very large amounts of data. This is because real-time data is often kept for 'historical' as well as backup reasons. Maintaining a regular 'snapshot' of the state of the data entities allows the history of the controlled environment to be considered. In addition, training exercises can take place using this history information. Typical storage sizes are given in [LC85]; an example is the US Coast Guard Vehicle Tracking System that can have 52Mbytes of live information at any one time. In non real-time systems, there is typically more data. In these systems, there is often more current, live, data than in a real-time system and copies are periodically backed up for security purposes.

Real-time data is often used only in predefined and limited ways. The user examining a set of data may only be able to access it in certain ways perhaps using a 'query-by-forms' technique of extracting the required data from the stored database. The reason for this is that providing the user with a sophisticated query language for manipulating the database introduces a great deal of processing overhead that often cannot be tolerated in the real-time environment. Providing limited access mechanisms means that these can be fine tuned at system design time to get the best possible performance out of the system. Compare this situation with a non real-time data processing application. For example, the database management system in a payroll processing system may provide the user with a sophisticated, run-time interpreted 'query' language such as SQL. The user of such a system can prepare programs in the query language, to access the database exactly to his or her requirements.

2.3 The Transaction Model

The transaction model can be the basis for the construction of reliable, fault tolerant computer systems [Mul89]. This section discusses the transaction model and describes two typical methods for its implemen-

[2] In a real payroll system, an uptime percentage of 75% is probably a little low. Non real-time designers still strive for as high availability as possible, although this is typically not as high as the availability requirements for real-time data.

tation. In addition, extensions for nested and distributed transactions are considered briefly.

2.3.1 Definition

A transaction is a collection of operations grouped together between a start transaction marker and an end transaction marker. The transaction reduces the attention the programmer must pay to concurrency control and failures by providing three properties [Gra78],[PBG87], [BW89],[Spe89],[Lis85] :-

1. failure atomicity.
2. permanence of results.
3. serialisability.

Failure atomicity ensures that if a transaction is interrupted by some hardware failure, then the partially completed work of the transaction is undone. The transaction can then be restarted when the fault is repaired.

If a transaction completes successfully, then the results of its operations are never lost and those results are made available to all other transactions.

Serialisability ensures that even though transactions may execute concurrently, their results are the same as some serial execution of the set of transactions. Serialisability ensures that concurrently executing transactions cannot observe the partial, perhaps inconsistent, results of other transactions.

We now refine the definition of a transaction as being a collection of operations bracketed by start and end transaction markers. Each transaction may be represented by the triple:

(R,P,W)

The R field is the read set of the transaction. This is the data that is used as input to the transaction. In a real-time system, this input may come from either the systems database or from some physical sensor reading data from the environment being controlled. The P field defines the set of operations, or processing, that must be carried out on the input data. A characteristic of the transactions model is that the processing does not have a state which is 'remembered' between

invocations of the transaction. (For a state based system where the processing to be carried out depends not only on R but also on previous invocations of the same transaction, then the previous state needs to be saved in the database. This state can then be read in as part of R.) W represents the results of the transaction, or the write set. In a real-time system, W may be written either to the system database or to some physical device 'controlling' the environment. Each transaction may be considered as a 'function' that has a set of input parameters and yields one result (i.e. W is an update to one database entity). On completion a transaction, can either abort or commit. If the transaction aborts, all its work is lost. If the transaction commits then the results are made permanent. Some commit protocol is required to ensure that future transactions can access the results of the committed transaction [Gra78].

2.3.2 Nested and Distributed Transactions

The basic transaction model just described can be extended to better support the parallelism of a distributed system and limit the effects of failures. Two typical extensions provide nested and distributed transactions. In the nested transaction concept, a transaction may 'spawn' multiple child transactions before it has completed itself. All the child transactions may execute in parallel. The net outcome of these parallel sub-transactions is the same as if they had executed sequentially. Each subtransaction can either commit or abort. If the sub-transaction aborts, the parent transaction should detect this and perhaps complete the work in some other way. If the sub-transaction commits, then its write set should not be made available to transactions outside the parent until the parent itself has committed. If the parent aborts then all results of completed sub-transactions should be lost. This concept may be altered slightly to encompass a network of transactions whereby on committing a transaction, a number of child or successor, transactions are created. In this extension, committing the child transactions does ensure their permanence.

The distributed transaction concept allows a transaction to reference data that is not stored at the place where the transaction was initiated. A set of remote sub-transactions are executed on the remote nodes where the data is stored. The sub-transactions are executed autonomously, out of the control of the initiating transaction. However, if the distributed transactions are nested in a parent transaction, then aborting the parent should also cause the nested, distributed, transactions to be aborted.

2.3.3 Implementation of the Transaction Model

Although the three properties of the transaction concept are equally important, the control of the concurrency within a distributed transaction processing system is often the first aspect to be considered. This section describes two related implementation approaches under the heading of the Client/Server model. Following this is a very brief description of how transactions might commit and how rollback and recovery can be implemented.

Client/Server Model For Concurrency Control

For reasons of security, abstraction and maintenance, data entities are often contained within and controlled by, protective subsystems or servers [Spe89]. There are then several ways in which the data can be accessed from these servers. Examples of these are the protected procedure call used in Multics [Sal74], capabilities [Fab74] and the client server model. In the client server model, each data entity is managed by a server that defines the operations that may be used by the client processes. A remote procedure call interface can then be used to invoke these operations.

The primary function of the data server is then to ensure that conflicting operations on the protected data entity do not occur. A description of the types of problems that occur if conflicting transactions are not controlled is beyond the scope of this chapter. For a summary of the need for concurrency control and the accompanying serialisability theory, see [PBG87], [BS79], [BG81], [KET76], [Pap79], [AT88], [TIM87].

Distributed Approaches

In distributed approaches to the client/server model there are multiple servers and transactions are directed towards these. The server considers the operations required on the data, and providing no conflicting operation is being executed, the operations of the transaction proceed. The server is structured as an infinite loop that continually receives transactions to be executed on the protected data entities. The server may have multiple threads so as to allow some transactions, such as multiple reads, to execute concurrently.

Centralised Approaches

In a centralised approach, as is often used in conventional database management systems, there is a central server that controls access to all data entities. For each data access, this central server is first consulted to check whether the data entity is available for the level of access

required. The main problem with the centralised approach is that the server can become a serious system bottleneck.

Implementing the Server

The are many ways in which the server can ensure that concurrent transactions cannot observe the partial effects of other, as yet incomplete, transactions. The serialisability guarantee of the transaction concept is often implemented using the conventional two phase locking protocol [PBG87], [KET76], [Gra78], [Men79], [Wol87]. Each server maintains a lock table for the data entities that it controls. This lock table records the current use of a transaction. When a new transaction wishes to gain access to an entity, the lock for that entity is consulted. If the required access does not conflict with the current access on the entity, then the transaction may proceed. If the required access does conflict then the new transaction is blocked until the current access on the data entity is released. The locking mechanism is known as 'two phase' locking because the transaction proceeds in two phases. In the first, the transaction obtains all the locks that it requires. In the second phase the transaction uses the data and releases the locks. In order to prevent deadlock problems, if a transaction is waiting for some entity, then it must release all the locks it currently has and start afresh. In addition, a transaction must not request any new locks after it has released a single lock.

A complementary approach allows each transaction access to the required data entities without any control. Problems of consistency are then dealt with when the transaction commits. This is called 'optimistic approach' lets each transaction execute as soon as is it submitted to the server. When the transaction commits, the server checks to see if the data entities had changed since the commiting transaction read them. If they have then the transaction is backed off and restarted. Optimistic approaches are often implemented by timestamping the transactions; this requires a global knowledge of time. [KR81] describes optimistic concurrency control in some detail.

Commiting, Rollback and Recovery

As with the concurrency control itself, there are numerous methods of committing a transaction. If there is only one copy of any data entity and a centralised server approach is used then the commit is simple. Any changes the transaction made to data entities are fixed and the locks on these entities released. If there is more than one copy of a data entity on multiple sites, the situation becomes more

complex. A simple method to commit transactions in a replicated data environment is attibuted to [Gra78], the two phase commit. Each copy of a data entity has a controlling server, the server that was consulted by a committing transaction is known as the coordinator. The coordinator sends a 'prepare to commit' request to each other server (subordinates). If these do not hold locks on the data entity, they are placed in commit mode and they reply to the coordinator with a confirmation message. If the coordinator receives confirmation messages from each subordinate, it commits the transaction and applies the updates made locally and remotely by sending an update propagation to each subordinate. On receipt of the update, the subordinate leaves commit mode.

If any subordinate server, in response to the prepare to commit message, replies with a denial message, then the coordinator server aborts the transaction and re-submits it at a later time. If a transaction was aborted at the commit stage then some other transaction was possibly also trying to commit at the same time.

There are many situations when a transaction needs to abort. An example is when some other transaction has committed and updated some data that the first transaction was using. Rollback is then the process of undoing the effects of a transaction. The simplest way of doing this is to record the database entity state before the entity is updated. Should the transaction abort halfway through, this copy can be reinstated. The technique called write ahead logging uses this simple approach. Other approaches to recovery also ensure that a transaction is both permanent and atomic. [Lam81] describes the concept of intentions lists which can be used to guarantee atomicity and permanence. Every change that a transaction wants to make to the database is stored in an intentions list for the transaction. This list is saved in non-volatile storage. If the transaction commits successfully, then the updates are made to the data entities in non-volatile storage. The intentions list is then deleted. If some part of the system fails before the commit is complete, then the intentions list can be used to finish the commit when the system is functioning again.

2.4 Failure of Commercial DBMSs in Real-Time Applications

This section comments on some of the more general problems of using an existing database management system for an implementation of the transaction model in a real-time system. The section is split into two

subsections. The first deals with commercial non real-time DBMSs and the second deals with database management systems specifically designed for real-time use.

2.4.1 Non-real time DBMSs

Most commerically available, off the shelf, distributed and centralised database management systems use the transaction as a basis for an execution model. There are, however, several reasons why these products are not generally well suited to real-time applications. Among these reasons are :-

- Response times. It has been estimated that real-time systems have much higher transaction rates in comparison with non real-time systems [Dix88b]. The smaller the response time, the more transactions can be processed in each time period. Typical transaction rates of up to 1000 simple database updates each second may have to be dealt with. Very few commercial DBMS products can claim to successfully match these requirements. [Dix88b] quotes that Oracle running on a MicroVax can process 20 update transactions each second. This is not nearly enough to be of use in many real-time environments.

- Excess functionality. Many commercial DBMS products provide excess functionality. Traditional real-time systems have fairly rigid requirements and, often, all database accesses are through standard queries rather than through the use of a complex and powerful query langage. Excess functionality implies additional and intolerable overheads in transaction processing.

- Generally lower resilience to failure. As stated in the section on the characteristics of real-time data, commercial, non real-time database products generally do not provide the resilience to failure that is required in real-time systems. Commercial DBMSs do not generally guarantee an 'uptime' approaching 100%.

- Closed architecture. Commercial DBMS products are generally based on a closed architecture. To extend the DBMS would require great involvement from the DBMS manufacturer. This could be costly, especially for the real-time system which is to develop and extend in the future.

- Non-determinism in processing times. The reason for non determinism in the processing times of transactions executed by

a commercial DBMS is explained in the next section. There is no guarantee that a given transaction completes within a certain time. This guarantee is of utmost importance in a hard real-time system.

2.4.2 Real-Time Specific DBMSs

There are several real-time specific database management systems that are available commerically. Among these is the Ferranti Relational Processor (DVME-785) [Dix88b], [Dix88a] and the Software Sciences Ltd, Diomedes Distributed Database Product [Law88]. These hardware database products aim to alleviate the problems suffered by conventional database management systems in real-time applications. The most important differences from conventional DBMSs are in the areas of response time, resilience to failure and expandability. [Dix87] quotes the Ferranti Relational Processor as processing 3000 transactions per second compared with the 20 tps achieved using Oracle on a MicroVax. No performance figures were available for the Diomedes product.

Both the Relational Processor and the Diomedes product can be configured in distributed systems. Many Relational Processors can be connected using the VME bus. The Diomedes product is based on a Transputer and as such incorporates the transputer's ease of construction of distributed systems. These products claim to provide better reliability and availability of data through distribution and replication.

A problem still arises with these real-time specific database products and that is the non-determinism in the execution time of transactions. Although the Ferranti Relational Processor claims to partly tackle the problem (search times of data are independent of the size of the searched data set) there are still elements of non-determinism. This is inherent in the concept of the transaction model and is considered in the next section.

2.5 Non-determinism in the model

There are some aspects of the transaction model that make it near impossible to guarantee with 100% certainty that a particular transaction is executed before a given deadline. This is referred to as the non-determinism of the transaction model. For a real-time system with very specific, hard deadlines for tasks, this non-determinism is a serious problem. This section considers these non-deterministic aspects

and discusses what can be done to alleviate them.

Communications

In a distributed system there is always communication, of one sort or another, between the connected nodes. The time it takes for a message to flow between two nodes (latency) depends on the load on the communications channel between the nodes. Knowledge of worst case latency is needed to guarantee that deadlines are met.

A partial solution to the non-determinism of the network latency cuts down on the amount of communication that is actually required, for example, by having transactions 'sited' at the same nodes as the data that they require and only replicating data for resilience reasons.

Disk Accesses

Similar to the problem of the non-determinism of the communications latency is the latency of accessing data entities stored on magnetic disk media. Disk latency is dependent on the state of the disk at the time the access is required. To reduce this non-determinism, we could remove the disk completely and introduce a main memory database architecture [Eic89]. The latency of access to main memory is less than that of disk storage and so the determinism of the transaction is improved. The main problem that then exists is how to ensure that the data in the main memory database is backed up to non-volatile disk storage for security reasons. This problem is beginning to be addressed [AJ89].

Concurrency Control

In a distributed database system, where there is concurrent access to shared, replicated, data entities, the major source of non-determinism is the concurrency control protocol. Whether or not a transaction is granted immediate access to a data entity depends on influences outside the transaction itself. The access is granted provided no other transaction is currently using the data entity. The problem is that at system design time there is no way of knowing what transactions are going to be executed at what time; the transactions are executed in response to external stimuli beyond the control of the system designer.

For some situations, an optimistic approach to the concurrency control may be sufficient. Transactions are allowed to execute on their local copies of a shared data entity and conflicts are sorted out at a later date. As an example suppose a transaction changes some part of the environment based on some shared and replicated data entity that it has read. Suppose also that another transaction changes the shared data entity during the lifetime of the first. In an optimistic approach, both these conflicting transactions execute concurrently and when both

are complete we decide what to do. In this example we may backoff the first transaction and restart it. Backing off the transaction has no effect on the database since it didn't update any data entities; restarting the transaction means that the latest copy of the data entity is applied to the external environment.

For most situations however, a strict concurrency control protocol is needed to ensure the continuous consistency of the database. Optimistic approaches are worse than locking for example when considering the determinism of a transaction. In optimistic approaches we may have to completely re-execute a given transaction; for locking based approaches we can have non- preemptive transactions in which once a transaction has started it runs through to completion.

2.6 Alternatives to the Transaction Model

Using the transaction model means that we have a data driven design; we consider the system from the transformations that are required on the real-time data entities. An alternative approach would be to use a process model. In the process model we specify explicity the steps to transform data from a triggering event through to the stimulus event. Each process can be considered a program that defines explicity the steps needed to transform the data and the order in which they are required. These programs may treat the shared data in the same way that a program treats local data but enclose access to the data within traditional program critical sections markers. The program requests the use of the data, and on completion relinquishes control of the data. The disadvantages of this approach compared with the transaction model are that the application programs have to explicitly consider the orderings of operations on shared data in order to preserve consistency; the concept of concurrency control is not implicit and manipulation of locks, or semaphores needs to be handled within the model and finally the application program needs to be aware of the problems of process failure and leaving shared data in inconsistent states.

2.7 A Proposal

The benefits of the transaction model are obvious. The transaction provides a recoverable, serialisable and permanent execution environment where the programmer does not need to consider the control of shared resources. However, as stated in the previous sections, the concurrency

control protocol of a real-time system has a serious effect on all timing aspects of the the application. It is important to realise these effects and at best remove them, but more likely, attempt to reduce them.

[Sta88] states that :-

> The fundamental challenge of real-time databases seems to be the creation of a unified theory that will provide us with a real-time concurrency control protocol that maximises both concurrency and resource utilization subject to three constraints at the same time: data consistency, transaction correctness, and transaction deadlines.

We therefore propose that in designing real-time database systems with a significant shared data content and hard timing constraints on the execution of transactions, both the timing constraints and the effects of concurrency control must be considered.

The following chapters, while not providing a unified theory of concurrency control and real-time scheduling as requested by [Sta88], nevertheless describe a methodology and supporting execution environment for the development of real-time database applications. This methodology considers the effects of concurrency control on the temporal aspects of the system from the first stages of the system design.

Chapter 3

A Model for a Real-Time System

Mercury, the Winged Messenger

3.1 Introduction

3.1.1 The Need for a Model

A requirements specification, derived from a systems analysis states what a systems must do. This is often expressed as a model. The system design methodology then describes how to progress from a statement of the problem in terms of some requirements specification, to the design of the system that conforms to the model. Before developing a real-time design methodology we need to specify the model that the resulting system will conform to.

By specifying this model, we can identify its parameters. The system design methodology can then be tailored to generating these parameters from the application specification. Most system design methodologies present a system model on which the methodology is based. Some describe the model in detail, others present a rather imprecise model.

3.2 A General System Model

3.2.1 Summary of the Model

A system consists of a set of tasks. Each task is independent and has a separate, identifiable trigger. Tasks may be synchronous or asynchronous with respect to each other. A task will have access to the system database, which is a set of data entities. Communication between two tasks is via any shared data entities. There is no direct message passing communication between two tasks.

A task instance is a particular triggering of a task. A task instance will use the latest versions of appropriate data entities. These versions are given increasing version numbers. On completion of a task, new versions of any updated entities are generated. Only one task may update a given data entity at any one time. This serialisation of updates is necessary to ensure that the integrity and consistency of the data entities is preserved.

A task consists of a number of transactions, ordered by a thread of control within the task. Transactions may read any number of data entities and, optionally, update a single data entity. The thread of control within a task is necessary to serialise conflicting transactions.

The use a task makes of a data entity may be described by a critical region. This represents the duration of the entities use. Within a critical region, the entity may be updated many times by the task. These updates will not be visible outside the task. When the task has completed, the final state of the entities that are updated by the task are made available to the rest of the tasks. These final states comprise the next versions of the entities.

3.2.2 A Task View

A computing system consists of a set of tasks. The definition of a task is as follows.

Definition 3.1 (TASK) *A task is that processing, data and control required in response to a* **single** *trigger from a source outside the task.*

The definition of a task is hierarchical. That is to say that a task is composed of subtasks, and these in turn are also composed of subtasks, where the definition of a subtask is the same as that of a task.

A task may be decomposed into serial, synchronous subtasks. In these, when a subtask has completed, it will send a control signal (triggering event) to the subtask that follows it. Any subtask B, that waits for a trigger from some other subtask A, is said to be a 'successor' of A; A is said to be a 'predecessor' of B. Besides serial, synchronous subtasks there may be parallel, asynchronous subtasks. With these, a task will send multiple control signals to each of the parallel subtasks that follow it i.e. to each of its successors. These will then execute asynchronously with respect to each other. When all of these parallel, asynchronous subtasks have completed, a single subtask is often needed. This subtask will be the successor of all the previous parallel tasks. Consequently, it is not triggered until all the predecessors have completed. In order to maintain the definition of a task being executed in response to a single trigger, there will be a 'merge' of triggers from parallel subtasks to trigger the common successor subtask.

The response triggers from tasks are optional. A task may or may not have any successor tasks to trigger on its completion. A task may also have a conditional triggering response. With this, the task will send triggers to a subset of its asynchronous successor tasks.

A real-time system consists of a set of special tasks. These tasks conform to definition 3.1 in addition to the following defintion.

Definition 3.2 (REAL-TIME TASK) *A real-time task is a task that has a trigger that originates in the controlled or monitored external environment.*

Real-time tasks are asynchronous with respect to each other. A real-time task has a single trigger which represents some event in the real-world. For example, an event in a chemical control plant may be a temperature reaching a critical state. The controlling computer system has an associated task to handle this situation. The task is executed when it receives a trigger to indicate the criticality of the temperature. Any communication between real-time tasks is through a shared data entity. This is similar to the State Vector Inspection of JSD [Jac83], [Sut88], but the communication can be bi-directional and the 'sending' real-time task has no knowledge of the state of the receiving real-time task. The only relationship, or coupling, between two tasks is that they can use the same shared data entities.

A task instance is a particular triggering of a task from the set of tasks that make up the system. A task instance is always given access to the latest versions of the data entities that it requires. If the latest version of the data entity is version n, then on completion of a task

Figure 3.1: High level representation of a task

that updates the entity, version n+1 will be available to other tasks.

Pictorial Representation of Tasks

A task may be represented pictorially by a box. Flowing into the box we have a trigger and leaving the box we have an optional response. This is shown in figure 3.1. This represents the highest

level of description of a task. In real-time tasks, this trigger will be from the external environment. The diagrams used to illustrate the nature of a task only consider the triggering and control flow through a task. The processing and data flow are not considered for the moment. Figure 3.2 shows the same task decomposed into three serial subtasks numbered 1.1 through 1.3. Each subtask is represented as a box. The enclosing 'dotted' box shows that these subtasks were decomposed from some larger task. The response from subtask 1.1 acts as the trigger to task 1.2; the response from task 1.2 acts as the trigger to task 1.3. These subtasks therefore represent serial, synchronous tasks. Figure 3.3 shows the same task but with subtask 1.2 further decomposed into two parallel, asynchronous subtasks. When subtask 1.1 finishes it triggers both subtasks 1.2.1 and 1.2.2 at the same time. These subtasks then execute asynchronously to each other. Subtask

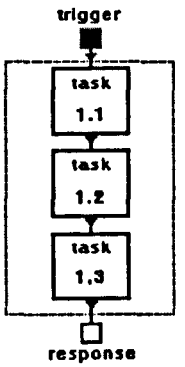

Figure 3.2: Decomposition into serial, synchronous subtasks

1.3 is executed when subtasks 1.2.1 and 1.2.2 have both finished. Figure 3.4 shows subtask 1.2.1 further broken into two serial, synchronous subtasks. Figure 3.5 shows a subtask 2 that on completion selects one of the serial subtasks 3 or 4 to execute. This conditional execution is represented by a 'dotted' line from the parent to each of the subtasks in the select. On completion of either subtask 3 or 4, subtask 5 may begin execution. This is represented by the 'merged' triggers leaving subtasks 3 and 4.

3.2.3 The Transaction

A task may be indefinitely decomposed into subtasks each with a finer grained description of the processing activities. However, there will come a point when there is no value to be obtained in further decomposition into subtasks. At this point, a task is decomposed into *transactions.* A transaction is defined by Definition 3.3.

Definition 3.3 (TRANSACTION) *A transaction is an atomic action that performs, at most, one update.*

A transaction is a set of transformations that can generate an update to a particular data entity. This set of transformations is not considered

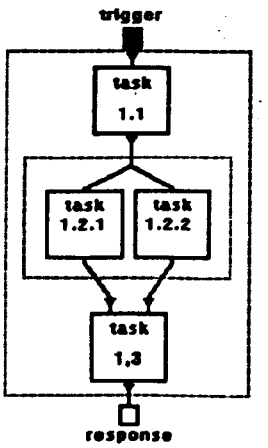

Figure 3.3: Decomposition into parallel, asynchronous subtasks

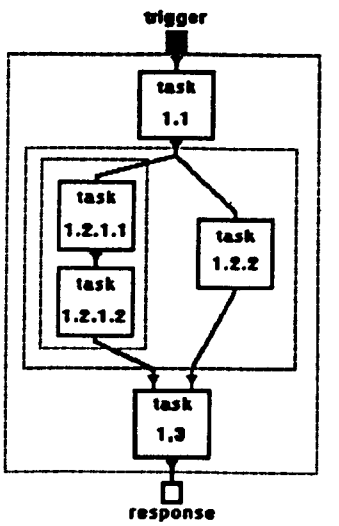

Figure 3.4: Further decomposition into serial subtasks

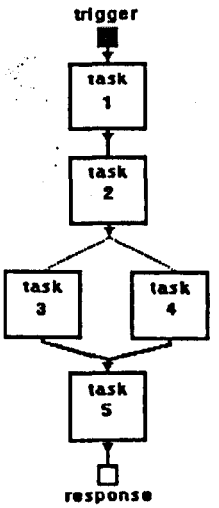

Figure 3.5: A select of one of two subtasks

to have any flow of control visible from outside. In an implementation, however, the transaction may have some control flow within it. This control flow is not recognised outside the transaction. A transaction is an atomic piece of processing. Each transaction will take a known time to execute. If the internal processing within the transaction requires iteration then maximum bounds on the number of iterations must be specified.

The execution of a transaction is controlled by a set of pre-conditions. These pre-conditions represent control flow from each of the other transactions that must complete before this one may begin execution. When a transaction has completed, then a control flow token can be sent to each member of its post-conditions. The post-conditions represents each transaction that waits for this one to complete before it may begin execution.

Controlling the Order of Execution of Transactions

The set of transactions within a task is ordered by the use that each transaction makes of the data entities used by that task. For example, suppose two transactions each use the same data entity and one of these transactions updates the data entity. To preserve the consistency of the

data entity and avoid the common problems found in concurrent access to shared data, the two transactions must be executed serially. Control flow within a task is dictated by such *data dependencies* (definition 3.4) between transactions. Where transactions have a data dependency they must be serialised. Where the transactions have no such dependency, they may execute concurrently with no control flow between them. It is the data dependencies that define the internal structure of a task in terms of serial and parallel subtasks. Serial subtasks can be used when there are data dependencies between the components. Parallel subtasks are possible when there are no dependencies.

Definition 3.4 (DATA DEPENDENCY) *A data dependency exists between two transactions if either they both update the same data entity or one of them reads the entity and the other writes the entity.*

Where two transactions have a data dependency, unless otherwise stated, the order in which the two transactions are serialised is arbitrary. In some circumstances, a task may be allowed to use 'stale' or out of date data. This implies that the task is using the data either at the same time as some other task is generating a more up to date version of the data or after another task has generated the new version of the data but before this version has reached the first task. It is important that the task that uses stale data does not generate a new version of that data since this 'illegal' new version is based on old data. Tasks that use stale data typically do not update the real-time database.

We distinguish between two sorts of transactions within a task. There are those transactions that transform one state of the internal database into another state. There are also those transactions that directly 'communicate' with the outside world via i/o devices such as consoles, sensors and actuators. The first type of transaction is the 'invisible' transaction; its effect are not immediately obvious to its environment. The second type is the 'visible' transaction; its effects are immediately obvious to its environment.

Pictorial Representation of Transactions

A transaction is a special instance of a task. As such, we can represent the transactions of a task in much the same way as we represent tasks themselves. Figure 3.6 shows a transaction representation of the task shown in figure 3.4. The transactions are drawn as circles instead of boxes. In the diagram control flow proceeds from the top to the bottom of the diagram unless explicitly shown with arrows. Each transaction

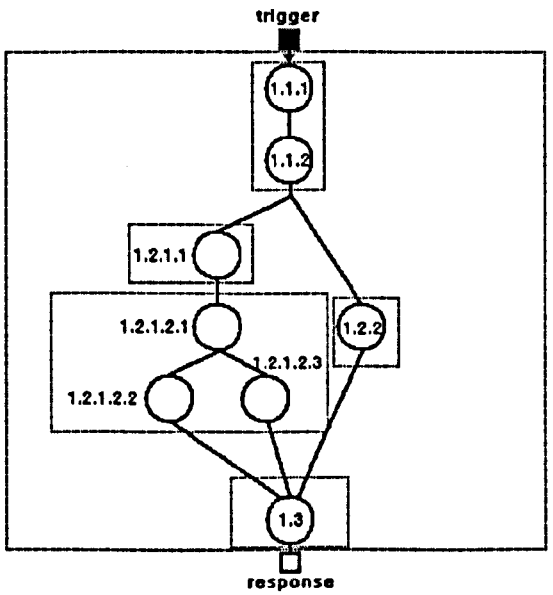

Figure 3.6: A Transaction Precedence Graph

is given a unique number based on the decomposition of higher level modules used to derive the transaction. A particular transaction is unique to the parent task. We decomposed subtask 1.2.1.2 of figure 3.4 into the three transactions 1.2.1.2.1, 1.2.1.2.2, and 1.2.1.2.3. Transactions 1.2.1.2.2 and 1.2.1.2.3 have no data dependency and can execute concurrently. Transaction 1.2.1.2.1 has a data dependency with both 1.2.1.2.2 and 1.2.1.2.3 and as a consequence is serialised with these two.

A representation of the transactions of a task in a graphical form as shown in figure 3.6 is known as a *Transaction Precedence Graph.* The graph shows the flow of control through the task necessary to ensure that each transaction is presented with and leaves, a consistent state of any data entities used.

3.2.4 Critical Regions

The use a transaction makes of a data entity has been described. The use a task, or subtask, makes of a data entity is more complex. Within a task, there may be many transactions that use a particular data entity. For each such entity there will be a *critical region.* A critical region is defined by Definition 3.5.

Definition 3.5 (CRITICAL REGION) *A critical region on a data entity represents the duration of a task's use of the data entity. The critical region is delimited by the task's first use and last use of the data entity. A critical region represents indivisible use of the data entity. Other tasks are only permitted conflicting access to the data entity before or after the critical region.*

Critical regions need to be indivisible because any updates to the entity within the task (or subtask) represent partial results. The final update is the only one of significance outside the task (or subtask). The critical region needs to be indivisible to ensure that the entity remains consistent within the task (or subtask).

3.3 Real Time Aspects of the Model

The task/critical region/transaction model so far described can be used to model any computing system. The only reference to real-time aspects came with the introduction of a real-time task. The real-time task is a special type of task with a trigger from the external, controlled or monitored environment.

3.3.1 Ensuring Indivisibility of Critical Regions

The definition of a critical region states that a task should have indivisble use of its data entities. There are two ways in which a critical region can be made to appear indivisible to other tasks. The first is to block any other task from accessing the data entity during its critical region. The second is to allow a second task access to the data entity and backoff and restart the first critical region when the second task has completed its own use of the entity. Since other tasks are only permitted access to the results of the final write of a critical region, at the task level each data entity appears to change at most once. This view of a task is the case even though within a critical region there may be many writes and reads to the data entity.

Critical Region Relationships

A task does not always consist of a set of unconnected critical regions. Two critical regions become connected when a given transaction appears in both regions. For example figure 3.7 shows a task consisting

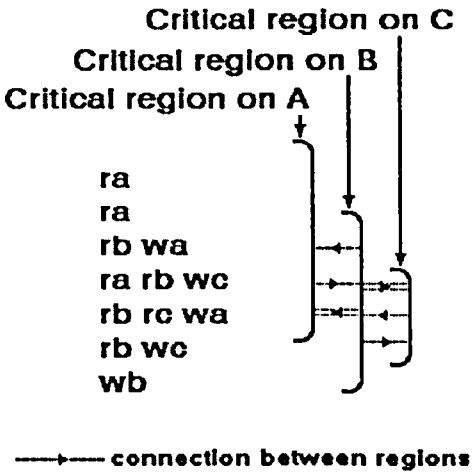

Figure 3.7: A Molecule (Connected Critical Regions)

of seven serial transactions. The task has three critical regions, on data entity 'a', data entity 'b' and data entity 'c'. These critical regions are connected because they share transactions. A connected critical region is known as a 'molecule'; the components of the molecule are connected, atomic critical regions. The critical region of 'a' is connected to that of 'b' through the third and fifth transactions. The region on 'a' is connected to that on 'c' by the fourth transaction. The region for 'c' is connected to those for 'a' and 'b' by the fourth transaction.

When backing off a critical region because some other task has changed the associated data entity, we need to consider any connected critical regions. For example, in the task of figure 3.7 suppose the fifth transaction had just completed when the task was interrupted by another which changed the value of entity 'b'. On restarting the sixth transaction, the value of 'b' has been changed by some external task. To ensure the consistency of the changed entity within the task, we have to backoff the task to the start of the critical region. This means we restart the task from transaction three. However, in the fifth transaction, the value of 'a' was affected by a now no longer valid value of 'b'. Consequently we should back off to the start of critical region 'a'.

In reality, molecules are made up of more than two atomic critical regions. The molecule will consist of many critical regions, woven to-

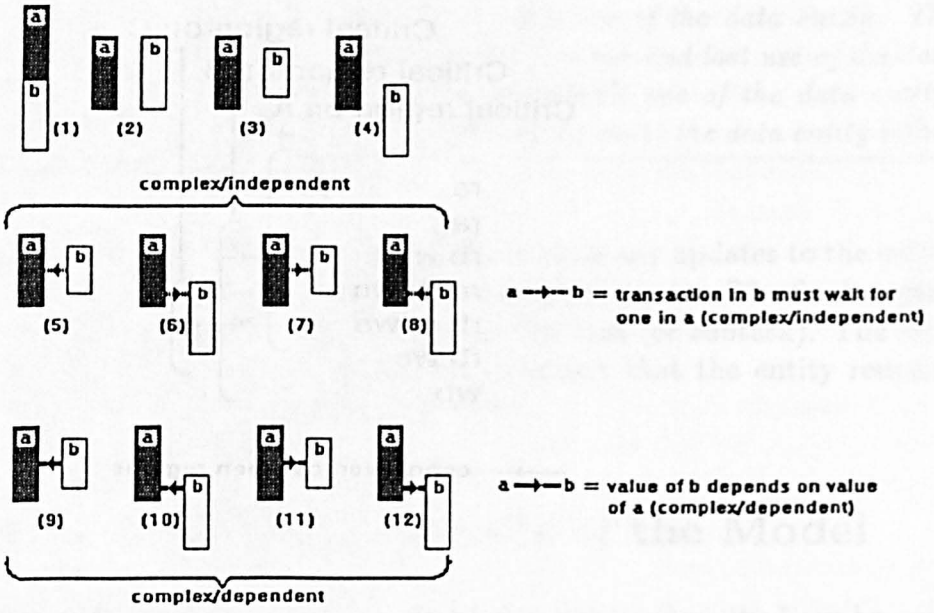

Figure 3.8: Relationships between two critical regions

gether by transactions that use more than one data entity. We need to know where to back off to, given that a particular critical region is being backed off. The process of determining this back off point is iterative. We first consider the interrupted critical region with each other critical region that it is connected to. Should any of these need to be backed off as a result, then we need to consider those other critical regions that these are connected to. This process continues until no more critical regions need to be backed of. The earliest point in the task that we have reached by successively backing off each critical region in turn if necessary, is the desired backoff point.

Figure 3.8 illustrates the possible combinations between two critical regions within a task. The combinations were generated by first of all considering the different 'overlappings' of the execution time of two critical regions. These were then extended to cover the case where there is a distinct relationship between these two regions. Figure 3.8 shows that there are twelve possible relationships between two critical regions. The first four are known as simple, or 'run-time', connectives. There are no shared transactions within the critical regions and no ordering constraints imposed on respective transactions by the application. The first four relationships represent the possible relationship between two critical regions based on their relative execution orderings. Relationship

1 represents sequential execution. Relationships 2,3 and 4 represent concurrent execution of the critical regions. If two critical regions are related according to any of the first four relationships, then we may back-off either of the two critical regions without affecting the execution of the other critical region.

Relationships 5 to 8 are known as the complex and independent critical region relationships. Although the critical regions in these relationships still do not share any common transactions (hence their independence) some application requirement specifies a constraint between their execution orderings that must be observed. The dotted arrow represents a transaction in one critical region that must wait for some transaction in the other critical region.

Relationships 9 to 12 are known as the complex and dependent critical region relationships. The bold arrow from 'a' to 'b' suggests that at some point in the two critical regions there will be a common transaction. This will read the current value of 'a' and use this to update the current value of 'b'.

We can now establish back off points for the relationships. We need to consider where to back off a critical region to given that the other critical region is being completely restarted. For relationships 1 to 4 we do not need to back off a critical region given that the other is being backed off. The critical regions are independent in all respects; neither has an influence on the other. For relationships 5 through to 12 we need to consider where to back off to given that both tasks are executing and one of them needs to back off. The results are summarised in Table 3.1.

Relation	Executing	Backing off	Back off executing CR to
5	a	b	first trans that waits for one in b
5	b	a	no back off
6	a	b	no back off
6	b	a	first trans that waits for one in a
7	a	b	no back off
7	b	a	first trans that waits for one in a
8	a	b	first trans that waits for one in b
8	b	a	no back off
9	a	b	first trans in a that uses b
9	b	a	no back off
10	a	b	first trans in a that uses b
10	b	a	no back off
11	a	b	no back off
11	b	a	first trans in b that uses a
12	a	b	no back off
12	b	a	first trans in b that uses a

Table 3.1 : Back off points for interruption of two critical regions.

In an implementation, there are two ways in which the problem of connected critical regions can be handled. In the first way, which has already been described, in backing off a critical region we can iteratively 'scan back' through the connected critical regions to find the primary back off point. In the example of figure 3.7 in backing off the critical region on 'b' we had to also back off that on 'a' to undo unwanted results.

An alternative approach is to recognise what entities are dependent on that critical region. At the start of this critical region, the state of these dependent entities is saved. In backing off the critical region, we restore the state of these entities. This avoids the necessity of having to back off any other critical region. In the example, at the start of the critical region on 'b', the state of entity 'a' is saved at the start of the critical region. If the region on 'b' is backed off then this state is restored, thus avoiding the need to repeat the first two transactions of the critical region on 'a'. The advantage of the state saving approach is that the minimum amount of work is backed off; the disadvantage is that large state saves are required.

3.3.2 Application Requirements Constraints

The order of execution of two transactions has so far been determined by the existence of a data dependency between the two transactions. There are some circumstances when there is a need to serialise several transactions even when there is no data dependency. This serialisation is known as the *Application Requirements Constraints (ARC)* and is defined in Definition 3.6.

Definition 3.6 (ARC) *An Application Requirements Constraint (ARC) represents the need to impose an ordering on two pieces of processing (transactions) where there is no data dependency between the two.*

As an example, suppose we have a task to shut down a chemical control plant and when this is complete, report this to the operators console. We may have a transaction to shut down the chemical processes and one to write to the operators console. Although these two share no data it is undesirable to have them executing concurrently; there is a danger that the operator will be informed of the system shutdown before the actual event.

Consequently we need an application requirement constraint between these two transactions.

It is the authors opinion that we only need application requirements constraints between visible transactions or between invisible transactions and visible transactions. The ARCs are needed to introduce subtle transaction orderings in the environment where there are no data dependencies between the transactions.

3.3.3 Introducing Timing Constraints

Central to the concept of a real-time system is the ability of the system to reason about and have a knowledge of time. In our model, tasks may be triggered at specific or arbitrary times. Triggered tasks have deadlines. A task will have a minimum time before it can be re-triggered. The real-time system has to decide what to to next when presented with a set of conflicting, triggered tasks. A later chapter describes how these decisions are based on the time properties of the tasks.

There are two recognised ways to represent time [Lam78], [LA90]. In the time point based representation (TPB) the view of the world is as a series of events that happen at some instant in time. The events take zero time to occur and result in a change in the state of the system.

The major disadvantage of the TPB representation is that events are not decomposable into sub-events such that an ordering can be imposed between these events [All83].

The alternative way to represent time is to use the time interval based representation (TIB). In this representation the view of the world is as activities that take a finite period of time to execute. Each activity will have an associated start and stop time. The TIB approach to time representation is more expressive than the TPB representation. It is easy to decompose an interval into sub-intervals and overlapping intervals can be expressed. A disadvantage of the interval based representation is that there may be a cummulative loss in time over a long period. This does not occur in the point based representation, and the time point representation can be used to represent intervals by expressing start and stop events.

In our model we need to be able to express and reason about both instantaneous events such as the triggering of tasks, as well as time intervals such as the minimum time between successive triggerings of a task. We thus need both types of time representation. Important times that are associated with each task are shown in Table 3.2.

Event/Activity	TIB or TPB	Comments
Trigger Time (S)	TPB	
Deadline	TIB	Start point for interval is S
Execution Time	TIB	Start point for interval is S
Min re-trigger time	TIB	Start point for interval is S
TIB = Time Interval Based, TPB = Time Point Based		

Table 3.2 : Time Represented Events and Activity

The trigger time for a task is represented by a TPB time. All other times are interval based relative to the trigger time.

3.4 Decomposing Applications to Conform to the Model

It is well recognised that there is a need to decompose large applications into more manageable units for detailed design [NS90], [BW89], [Par72].Often this decomposition is left to the intuition and experience of the systems designer. Some heuristics are often presented to guide the decomposition. Such heuristics are for example to split the application into subsystems such that the partitions contain activities with

similar timescales; or such that the partitions contain activities that are closely coupled to the external environment. A further heuristic is to group those activities such that communication across partition boundaries is minimized [Ben88].

We favour a more rigid approach to application decomposition hinted at in [NS90] and [YC78]. An initial decomposition is performed that is still intuitive. This decomposition splits the application into sub-systems. A sub-system contains all activities that are functionally related i.e. with a high degree of cohesion [YC78]. As an example, consider a chemical control plant with several controlled chemical processes. The initial decomposition may yield a sub-system for each of the chemical processes and a further one to handle operator interaction with the system. Each chemical process sub-system contains activities devoted to managing the respective chemical process. The operator sub-system contains activities relating only to the operator.

Sub-systems are still too large to be manageable. A further decomposition of each sub-system is requried. We use the real-time task concept to identify a set of tasks of a more manageable size. As an example, the chemical process sub-system could be decomposed into a task to handle extremes of temperature and another to handle extremes of pressure. Further decomposing real-time tasks into transactions and the design of the computer system to implement the tasks is described in the following chapter.

3.5 Modelling Tasks with Petri-nets

The transaction precedence graph (for example in figure 3.6 may be considered as a special case of a Petri-net [Pet77], [PS89], [Mur]. The transactions in the TPG represent the 'transitions' of the petri-net and the precedence constraints between the transactions in the TPG represent the 'places' of the petri-net. A transition (transaction) fires (executes) when each of its input places (pre- condition execution constraints) contains a token (is satisfied).

The petri-net model can then be used to guide an implementation of an executing transaction precedence graph. A transaction scheduler is responsible for counting the number of tokens each transaction has. When a tranaction has enough tokens the scheduler executes the transaction. On completion, the scheduler gives a token to each of those other transactions that was waiting for the first to complete.

There is a problem with t he anal ogy between transaction precedence

graphs and petri-nets. The petri-net only describes 'enablement' of a transition ie. the petri-net describes the conditions under which a transition is enabled. An enabled transition may then fire but the petri-net does not explain exactly when the firing takes place. The petri-net could be considered a visual representation of temporal logic [Har87], [Lam78]. If a transition is enabled then we know that all those transitions it depended upon had fired at some point in the past. If a transition is enabled then a token will eventually be passed to each of its output places at some point in the future; those transitions connected to these output places may then be fired. This model does not exactly describe the characteristics of the transaction precedence graph where a transaction is 'immediately' executed when each of its pre-conditions are met.

3.6 Conclusions

This chapter has presented a model of a real-time system. The real-time system is constructed from a set of tasks; each task has a separate and identifiable trigger. Real-time tasks are those tasks that a triggered by the occurrance of some event in the outside, controlled, world. Each task is constructed from a number of transactions. The ordering of transactions is defined by data dependencies and application requirement constraints. A complete ordering of the transactions within a task is depicted as a transaction precedence graph. The complete ordering of transactions contains a number of critical regions. Each critical region is associated with a particular data entity; the need to back off the work carried out in these critical regions was explained.

3.6.1 Glossary of Common Terms

We now present a summary of some common terms which are used later in the thesis.

- **TASK.** A task is that processing, data and control required in response to a single trigger from a source external to the task.

- **CRITICAL REGION.** A given task has a critical region for each data entity that the task uses. The critical region on a data entity represents the duration of the tasks use of the entity. Critical regions on different entities may overlap or be disjoint.

- **TRANSACTION.** A transaction is an atomic task that performs, at most, one update.

Chapter 4

Data Entity Viewpoint Analysis

Jupiter, the Bringer of Jollity

4.1 Introduction

The previous chapter introduced a model for a real-time database system. In this chapter we describe a design methodology that transforms the specification of a real-time system into a form consistent with this model. A series of steps to transform the specification of the system is described. In addition, a new notation is presented. Traditional notations consider the real-time system from the flow of data or flow of control viewpoint. The new notation presents the real-time system from a different aspect to that of traditional real-time design methodologies. This notation has many uses. These are described.

4.1.1 Motivation

The previous chapter described the composition of a real-time system from a set of tasks and transactions. This description considered the real-time system from a flow of control viewpoint. This viewpoint is also the basis for petri-net modelling of systems. A complementary viewpoint considers the flow of data through the elements of the system. This viewpoint is taken in approaches such as [YC78] and [Jac84]. A combined viewpoint considers the effect that data access has on the control flow of the real-time system.

In considering a single real-time task, a control or data flow approach is sufficient to model many aspects of the behaviour of the task. However, when many tasks are considered and where those tasks share common data, the control/data flow approaches fail to capture some important information. This information concerns how the tasks indirectly interact through constraints imposed on their access to shared data. A major source of non-determinism in a database system is the concurrency control protocol and its effects owing to the backing off or suspension of tasks wishing to access shared data. A fundamental challenge for real-time database system designers is to design database concurrency control protocols that ensure transactions meet deadlines as well as ensuring consistency and correctness [Sta88]. We propose that, by considering the concurrency control at design time, its effects at run-time can be determined.

This chapter considers the real-time database from a data entity viewpoint. The role that each data entity plays in the system is considered in this viewpoint. A diagrammatic notation is described to express the role of each data entity in the real-time database. Considering the real-time system from a data entity viewpoint has many advantages. This chapter describes these advantages and shows how the data entity viewpoint analysis is useful.

4.1.2 Objectives of the analysis method

The objectives of the Data Entity Viewpoint analysis are to describe the role that each data entity plays in the real-time system and to use this to consider the effects of concurrency control on the determinism of the real-time transactions. The data entity viewpoint analysis will be seen to underly the data/control flow viewpoints and is used with these to model all aspects of the real-time system.

The data entity viewpoint analysis can be used to generate a control flow viewpoint of the real-time tasks. That is, given a task set of transactions, their access requirements and some partial ordering that represents the application requirements constraints (ARCs), considering the transactions from a data entity viewpoint allows us to generate a transaction precedence graph. This complete ordering will maintain the partial ordering on the transactions imposed by the requirements of the application. In addition, this complete ordering of transactions will ensure the consistency of the database as well as the correctness of the transactions. Although the ordering is seen as complete, an order may not be expressed between some transactions. These can execute

concurrently.

The data entity viewpoint analysis can also be used to determine effective concurrency levels for each data entity. The concurrency level for a data entity is the maximum number copies of the entity that can be used at any one time. In generating the transaction precedence graph for a task, we can determine the worst case execution time for the task and permit the static scheduling analysis described in a later chapter.

4.1.3 Pre-requisites for the method

This section will describe the stages of systems analysis necessary before the data entity analysis can be carried out.

Step wise refinement to generate transactions sets

Step 1 : Identification and Definition of Real-Time Triggers
The first stage in decomposing a real-time application into a set of transactions is to identify the triggering events from the controlled environment. These triggers will cross the boundary between the external environment and the computer system. The triggers will typically come from external sensors. In addition, in this stage we must identify the actuators and devices through which the real-time system controls the environment. To represent these external influences we use a modified context diagram. Suppose we have a control system for a chemical vat. The vat has four sensors and two actuators associated with it. Two of the sensors are related to the temperature in the vat; one is triggered periodically and the other is triggered when the pressure is critical. The two other sensors are for the pressure in the vat. Again, one is triggered periodically and the other is triggered when the pressure in the vat goes critical. The two actuators affect the temperature and pressure in the vat. This system is represented by the context diagram shown in figure 4.1. Actuators are represented by circles and triggers by boxes. Those triggers marked with a tilde are the periodic triggers. Where there is no marking, the trigger is sporadic.

Step 2 : Decomposition of System into Subsystems
The second stage in the decomposition is to break up the real-time system into subsystems. The goals to be met in this decomposition are that the subsystems consist of modules that show a high degree of functional relatedness i.e. a high degree of 'cohesion' and that the 'coupling' between the subsystems is as low as possible. Figure 4.2 shows a subsystem decomposition diagram (SDD) for the chemical vat

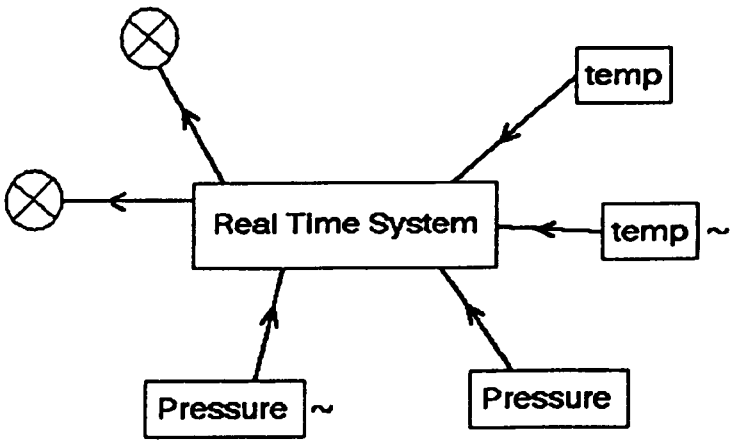

Figure 4.1: Context Diagram for a simple Chemical Control Plant

control system.

Step 3 : Decomposition of Subsystems into Tasks
The third stage in the decomposition process is to break up each subsystem into its constituent tasks. For each 'input' event to the subsystem, there will be one associated task. A task may or may not be associated with a number of the 'output' actuators. The task decomposition can be shown on a Task Decomposition Diagram (TDD). An example of this is shown in figure 4.3.

In some circumstances, a collection of processing activities should be activated in response to more than one event triggering. For example, in the chemical control plant, we may need to shut the chemical vat down if the temperature and the pressure are both triggered. To do this we can still have two separate tasks, one for the temperature trigger and the other for the pressure trigger. When the temperature triggers, it sets a flag to show that the temperature has gone critical. The task then tests to see if a similar flag has been set by the pressure task. If this second flag has been set then the task will shut down the vat. The associated pressure task will work in a similar way.

At this stage, the timing characteristics of each task need to be determined. For each task we need to known the minimum repeat time (or interval). This is the minimum time between consequetive triggerings

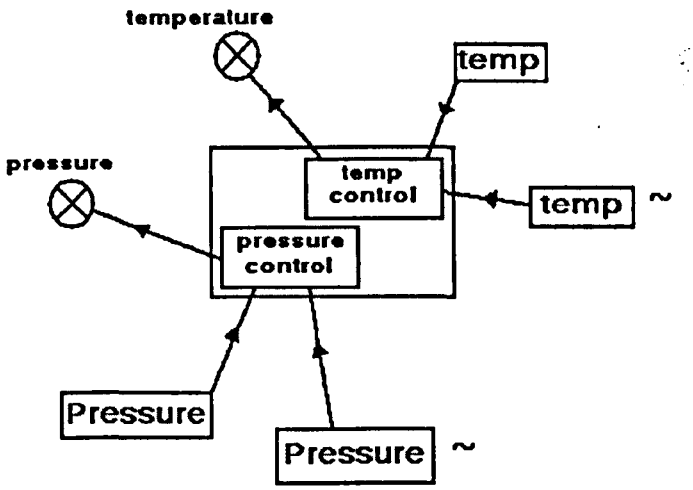

Figure 4.2: Subsystem decomposition diagram (SDD) for a chemical vat

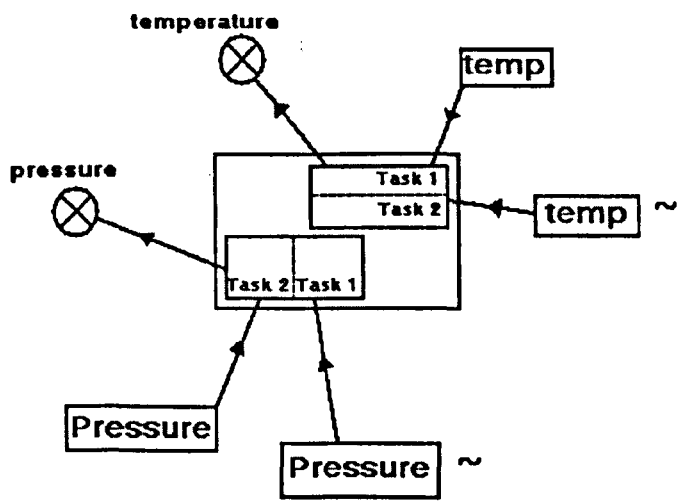

Figure 4.3: Task Decomposition Diagram a chemical vat

of the task. The minimum repeat time for a task will depend on the external environment and the peripheral hardware e.g. sensors that the task is associated with. For each task we also need to know the deadline. This is a relative value and is expressed as a maximum execution time allowed for the task.

Step 4 : Initial Design of Real-Time Database
In this stage we provide an idea of the final form of the database. It is not necessary to know the exact form of each of the data entities in terms of what fields and record structures will be used. Instead, we need to know the major shared data items that will be used by the real-time system. Refinement of the real-time database takes place at a later stage. For example, in a real-time radar tracking system, we would have a data entity to hold the current positions of all objects within the field of view of the radar. We may also have another data entity to record those objects that are on a potential collision path. At this stage, we do not need to know that the track table is made up of many records; each record being made up of a collection of fields.

Step 5 : Decomposition of Tasks into Transactions
The final decomposition stage is to generate a set of transactions for each task. Each transaction will take the form:

(Pre-conditions,Readset,Processing,Writeset,Post-conditions)

Each member of the read and write set is a data entity identified in the previous step of the decomposition method. The processing is that sequence of actions necessary to transform the readset into the writeset. The processing should be described as a series of transformations. The description of the processing corresponds to the 'minispec' of other design methodologies [NS90], [YC78]. How the transaction boundaries are identified is left to the intuition and experience of the designer. Some heuristics are available. A sensible, smallest 'grain' size for a transaction is that processing required to update one data entity. However, a data entity may be an object any size. Consequently, the size of the transaction is very much dependent on the size of the data entity that it updates. If the entity is, for example, a record in a table, the transaction could be very short. Alternatively, if the updated entity was a large, complex table, the transaction may correspondingly be large and complex. There are no 'hard and fast' rules that should be applied to the size of the transaction, it is dependent on the structure of the real-time database.

However, there are rules that can applied to ensure that the transactions

are of the best size to allow full concurrency within the set of transactions. In adopting a functional approach to viewing each transaction, the write set of the transaction consists of one member. Providing, the write set of one transaction does not conflict with the read sets of another transaction, these transactions may execute concurrently. Another sensible heuristic would be to ensure that the minispec for the transaction was no longer than half a side of A4 in length. Again this corresponds to heuristics in other design methodologies and ensures that the transactions are of a manageable length (although, again, this length is often dependent on the size of the entity that is accessed).

The pre-conditions are those transactions that must have completed before this one can begin execution. The post-conditions are those transactions that may execute when this transaction has completed. The pre- and post-conditions are used to implement any ordering constraints imposed by the application (i.e. the ARCs). For example in the chemical control plant shut down task of the previous section, if transaction 1 shut down the chemical vat and transaction 2 informed the operator of this action then transaction 2 would appear as a post-condition of transaction 1. Similarly, transaction 1 would appear as a pre-condition of transaction 2. By identifying the pre- and post- conditions for the transactions, we impose a partial ordering on the transaction set. The pre- and post- conditions are found from the ARCs

When all transactions for a task have been specified, iteration and selection constructs need to be identified. The maximum number of iterations needs to be found for later scheduling analysis. In addition, the time a transaction takes to execute should be determined at this stage. [Sho83] gives advice on how to estimate the size of a program. This can be used to help estimate the execution time by counting instructions. This stage represents a departure from a top-down approach to the systems design. [LM88] states that detailed program design is often necessary early on in the design of real-time systems to determine the feasibility of meeting deadlines. The timing information gathered at this stage is not necessary for generating a control flow viewpoint for each task. The information is however needed to verify the timing properties of the tasks. Where transactions contain loops and selection within the processing part (iteration/selection within a transaction), the worst case execution times should be determined. This requires an understanding of the application as well as some assumptions. For example suppose we have a transaction to do some statistical analysis on each member of a track table. We need to know the maximum size of the table to predict the worst case execution time for the transaction. The size of the table is connected to the maximum number of objects

that can appear in the radar scope at any one time.

On completion of this stage, we have a set of transactions each transforming some real-time database entity in some way. This situation is very similar to that of functional languages. We express what transformations we wish carried out on the real-time database (what we want done) and not, at this stage, how this is accomplished. The order of execution of the transactions is determined later. This 'functional' approach is different from the more traditional control flow approaches where the designer specifies both the actions to be taken and a complete ordering of these actions.

4.2 The New Notation - Data Dependency Rings

This section introduces a new notation for describing the role that the data entities play in the real-time system. The notation has two main functions. The first is as an aid for describing the behaviour of the system. When used with data/control/event flow diagrams, a complete picture of the behaviour of the system can be defined. The second use for the notation is as an intermediate stage in the automatic creation of a control flow representation of the real-time tasks.

The data dependency ring notation is a hierarchical, graphical notation that expresses the use that each task makes of each data entity in the real-time system. In addition, the ring notation describes the partial orderings of transactions in each task (ARCs) and any selection between successive transactions. These are all the known facts about the control flow within a task.

4.2.1 Why Introduce Another notation?

Existing diagrammatic notations consider the system from one of two viewpoints. The first is the flow of data viewpoint. The second is from the flow of control through a set of activities. Neither of these express the additional flow of control that is introduced at run time to control concurrent access to shared data. The new notation aims to show where this type of additional flow control is necessary.

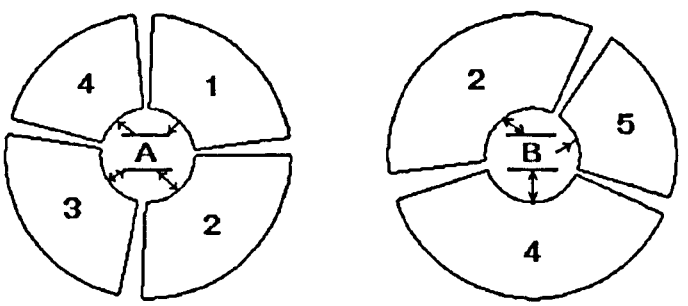

Figure 4.4: Primary DDRs for Two Data Entities

4.2.2 Structure of a ring

A ring notation describes the role of each data entity in the real-time system. Each ring is known as a Data Dependency Ring. The ring emphasises the closed and complete description of the data entity. The description of the data entity is placed in the middle of the ring. All tasks that use that data entity are then listed around the outside of the ring. Within these tasks, each transaction that uses the data entity is listed. The required type of access is also shown; reads, writes or reads and writes.

A primary ring is drawn for each data entity. The data entity name is shown at the centre of the ring and each task that uses that entity is listed on the edge of the ring. Figure 4.4 shows the primary rings for two data entities 'A' and 'B'. Entity 'A' is used by four tasks and entity 'B' is used by three tasks. Task 1 writes to entity 'A'. Task 2 reads and writes both entity 'A' and 'B'. The primary rings are useful to illustrate the need for control between concurrently executing tasks. In the example, tasks 1 and 5 do not share any data entities; there is no control needed between these two tasks i.e. they are have functional as well as data independence.

For each data entity, a secondary ring is drawn. This illustrates every

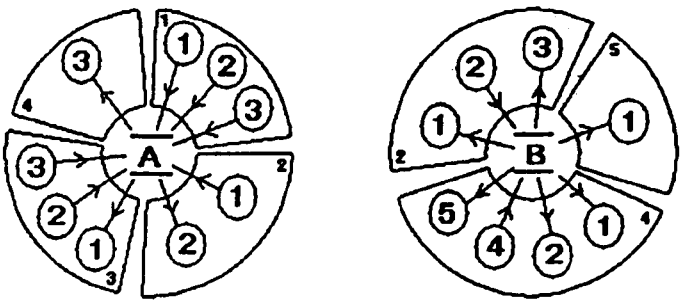

Figure 4.5: Secondary DDRs for Two Data Entities

transaction in every task that uses that entity. The transactions are enclosed within their own task bounds. Again, the access that the transaction requires is illustrated. If the transaction writes to the entity an arrow points towards the entity. An arrow of the opposite direction is used where the transaction reads the entity. Figure 4.5 shows the secondary rings for two entities of figure 4.4. These DDRs show, for example, that task 1 transaction 1 has a read set of 'B' and a write set of 'A'. The main use for the secondary ring is to illustrate where there is a need for control of concurrently executing transactions within a task.

Where a data entity can be decomposed into smaller units, tertiary DDRs may be drawn. For example, suppose data entity 'A' in the previous examples consisted of three parts each describing the physical characteristics of a particular chemical vat. The entity could be decomposed into three parts A', A" and A"'. Tertiary DDRs for these sub-entities could look like those in figure 4.6. This figure shows that A' is used by the four tasks whereas A" is used by tasks 1 and 3 only. The figure also shows that within task 1, the three transactions 1,2 and 3 can execute concurrently with no control.

The use of primary, secondary and then tertiary DDR shows the hierarchical nature of the notation. The primary DDRs show the concurrency

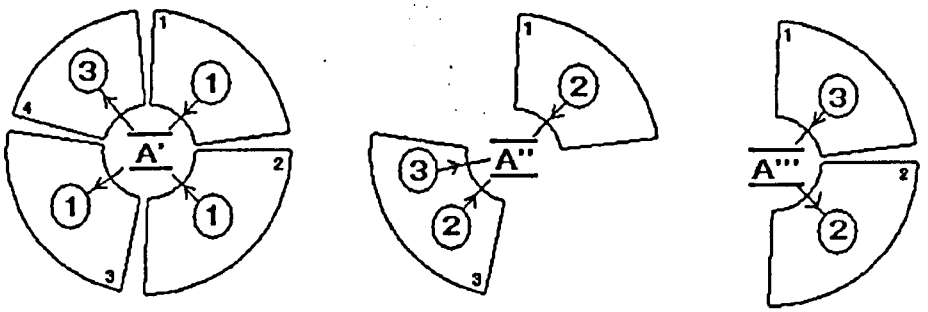

Figure 4.6: DDRs for A Decomposed Data Entity

at the task level. This is the number of tasks that can concurrently require access to the data entity. The secondary and tertiary DDRs show the concurrency within a task i.e. at the transaction level. The level to which tertiary DDRs are constructed is dependent on how well the entities can be decomposed into smaller units. Decreasing the 'grain' size of the entities may improve the concurrency that is available but it does increase the overhead of concurrency control that is needed [PBG87].

The description of the DDR presented so far ignores the control flow that the designer explicitly states in the design of a task. This control flow is the partial orderings of transactions (ARCs), iteration of transactions and selection between a choice of transactions. Partial ordering between transactions is expressed by listing next to a transaction, all those transactions that it waits for. An example of this is shown in figure 4.7. In this transaction 2 must wait for the completion of transaction 1.

Where a transaction must choose between one of many successor transactions, a dotted 'choice' line is drawn from the transaction to the outside of the ring. The 'arms' of this choice line then represent the transactions that are optionally executed. The marking at the end of an arm shows the selected transaction. An example of a selection is shown in figure 4.8. This DDR shows that on completion of transaction 1,

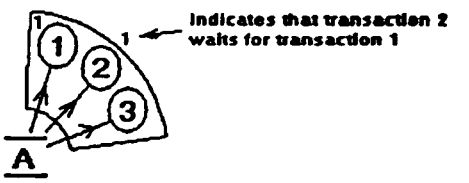

Figure 4.7: Expressing Partial Ordering on the DDR

either transaction 2 or 3 will be executed.

Iteration among transactions is expressed using the selection construct. The top of the loop and the first transaction after the end of the loop are listed as selection transactions of the bottom transaction in the loop. Iteraction can occur in three ways:

1. Iteration of tasks. This represents continuous retriggerings of a task and is not represented in the notation.

2. Iteration within a task i.e. at the transaction level. This could be for example where we have a number of transactions within a task being applied to a number of different parts of a data entity.

3. Iteration within a transaction. This iteration is invisible at the transaction level and is not expressed in the notation. It is however important to determine the worst case execution time of this sort of iteration.

4.3 A Simple Example

This section presents a simple example to demonstrate the steps necessary in transforming a specification of a system into the structures used

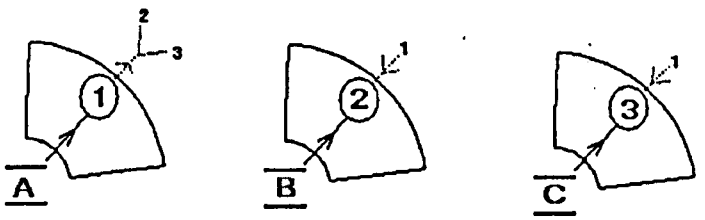

Figure 4.8: DDR Showing Selection

in an implementation. We will start with a simple, English, description of an Automated Bottling System. This system is adapted from the bottle-filling example of [WM86] and [NS90]. The steps necessary in decomposing this vague specification into tasks and then transactions are described. Finally the data dependency rings are constructed. From these, the transaction precedence graphs may be drawn. For this simple example, we go no further than constructing the data dependency rings; generation of TPGs and allocation schemes is left as an exercise.

4.3.1 The Bottling System

The system consists of bottle filling lines fed by a single vat containing the liquid to be bottled. The function of the control system is to control the level and the pH of liquid in the vat, to open and close valves to release liquid from the vat into the bottles, and to inform the human operators of the state of the system.

The vat level control is accomplished by a periodic read of a sensor in the vat and adjusting a liquid input valve accordingly. The pH value of the liquid needs to be monitored because the pH of the liquid changes over time. A constant pH is maintained by introducing a chemical that reverses the pH change so as to keep the pH at a constant level.

The amount of 'pH-leveling' chemical that is added depends on both the current pH of the liquid and the rate of flow of liquid through the tank. A sensor in the tank is used to measure the pH and this is read periodically. Should the pH go outside a predefined range, all control actions are suspended and the system shut down. The pH value of the system is then restored to a safe level manually.

Bottles are filled from the tank as follows:

- A bottle drops onto the filling platform depressing a bottle contact sensor. When this sensor is triggered, the bottle is in the correct position to be filled.
- The valve from the vat is opened and a measured amount of liquid flows into the bottle. The amount of liquid is controlled by measuring the weight of the bottle and its contents.
- When the weight reaches a predetermined value, a further sensor is depressed. The valve from the vat is closed and the bottle is moved across to the final stage of the bottling process (not considered in this example). Another bottle may now drop onto the filling flatform to repeat the process.

In addition to the actual filling of the bottles, the control system records the amount of liquid consumed, amount of bottles filled, volume of liquid in the tank and the current pH of the liquid. All this information is displayed on an operators console. The operator may close down the entire system by pressing the 'off' button on the console.

4.3.2 Subsystem Identification

This simple bottle filling control system has three distinct areas of control which are the separate subsystems from which the system is constructed. These subsystems are:

- Bottle fill subsystem. This is responsible for filling the bottles from the vat.
- Vat control subsystem. This is responsible for monitoring and controlling the volume and pH of the liquid in the vat.
- Operator subsystem. This is responible for keeping the operator informed of the state of the bottling line.

4.3.3 Identification of Real-Time Triggers

The real-time triggers are identified by considering the devices that the bottling system is connected to. The triggers are as follows:

- Off signal from the operator
- Bottle contact sensor indicates arrival of new bottle
- Bottle weight sensor indicates the bottle is full
- Clock trigger to periodically read the pH of the vat
- Clock trigger to periodically read the volume of liquid in the vat
- Clock trigger to periodically update the operators screen

The actuators that the control system can affect are as follows:

- Valve from the vat to the bottle (ACT1)
- Valve to control raw material entering the vat (ACT2)
- Valve to introduce pH levelling chemical (ACT3)
- Switch to turn off the whole system (ACT4)

The real-time subsystems, triggers and actuators can be seen in the subsystem decomposition diagram shown in figure 4.9. For each of the above triggers, there will be a corresponding task. These six tasks are described later.

4.3.4 Definition of System Data Entities

The implementation of the bottling system will use the following data entities:

- Bottling Plant Status (BPS). This will record the status of the plant (either 'off' or 'on'); the volume of the liquid in the vat, the pH of the liquid and the rate of flow of leveling liquid into the vat; the rate of flow of liquid from the vat and the number of bottles filled; and the rate of flow of raw liquid into the vat. This information is used to keep a record of the functioning of the system and supply the operator with up-to-date status information.

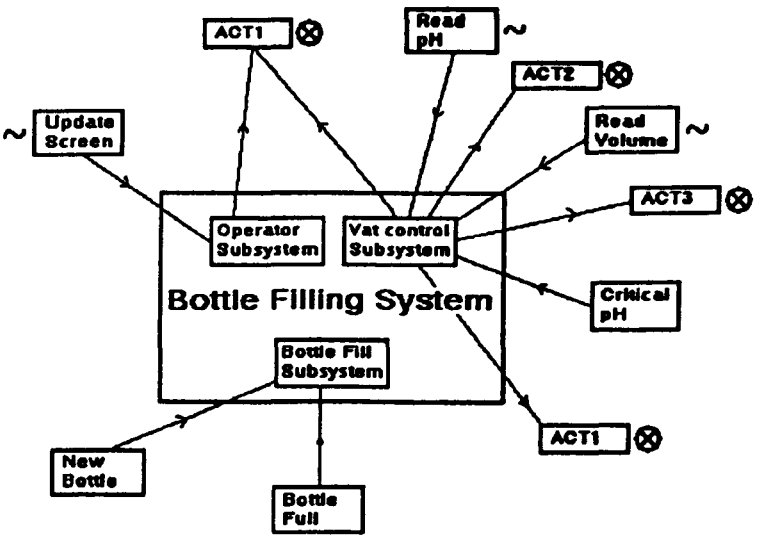

Figure 4.9: Subsystem decomposition diagram for the bottling plant

- Desired pH of liquid (DP).
- Desired volume of liquid in vat (DV).
- Current pH of liquid (CP).
- Current volume of liquid in vat (CV).
- Critical pH of liquid (CrP).

For simplicity and in order to maintain a uniform interface, the hardware actuators can be accessed as though they were database entities. The implementation of the system will differentiate between the two sorts of access. This is similar to the Unix philosophy of treating hardware devices as special files.

4.3.5 Description of task database usgage

This section describes each task in terms of actions on the data entities defined in the previous section.

Task 1 : Operator presses the Off button

When the operator presses the Off button on the console, the whole bottling system should be shut down. The following actions on the system actuators and data entities must take place.

1. Close the vat to bottle valve (ACT1).
2. Close the raw material to vat valve (ACT2).
3. Close the pH to vat valve (ACT3).
4. Update the BPS to indicate that there is no material flow.
5. Shut down the system power to the bottling line (ACT4).

Task 2 : New (empty) bottle ready to be filled

An empty bottle has dropped onto the bottle filling platform and triggered the sensor. The filling of this bottle should now commence.

1. Open the vat to bottle valve (ACT1).
2. Update the BPS to show the new rate of flow of liquid from the vat.

Task 3 : Bottle full

The bottle has reached the desired weight. The filling should now stop and the bottle should be moved off to the next part of the system.

1. Close the vat to bottle valve (ACT1).
2. Update the current volume of the vat to show that a bottle has been filled from it.
3. Update the BPS to indicate that no liquid flows from the vat and that another bottle is now full.

Task 4 : pH in tank goes critical

The critical pH sensor has triggered. The system should be shut down so that the operator's can manually restore the bottle filling line to a safe status.

1. Close the vat to bottle valve (ACT1).
2. Close the raw material to vat valve (ACT2).
3. Close the pH material to vat valve (ACT3).
4. Update the BPS to show the inactive state of the system.
5. Shut down the power to the bottling line (ACT4).

Task 5 : Monitor pH

The pH is monitored periodically. The function of this task is to adjust the amount of pH levelling liquid that enters the vat. The following actions are required.

1. Read the Desired ph (DP), the current volume of liquid (CV) and the current pH (CP). Work out the rate of flow of liquid through the tank and calculate a new value for the pH levelling liquid rate of flow. Change the pH to vat valve to alter the quantity of pH levelling liquid in the vat (ACT3).

Task 6 : Monitor level of liquid

The level of liquid in the vat is monitored periodically. The function of this task is to ensure that there is always at least a certain quantity in the vat. The following actions are required.

1. Read the current volume of the vat (CV). Read the desired volume of the vat (DV). Calculate the amount on liquid required in the vat and change the raw material to vat valve to reflect the required volume (ACT2). Update the new current volume (CV).

Task 7 : Update the Users Screen

The operator's screen should be refreshed periodically with information such as the number of bottles filled; the volume of liquid in the vat; the current pH of the vat. All this information is stored in the Bottling Plant Status (BPS) data entity. The function of this task is to take the information in the BPS and write it to the screen.

1. Read the BPS and update the operator's screen.

4.3.6 Data Dependency Diagrams

Figures 4.10 and 4.11 show the data dependency rings for the system data entities and actuators.

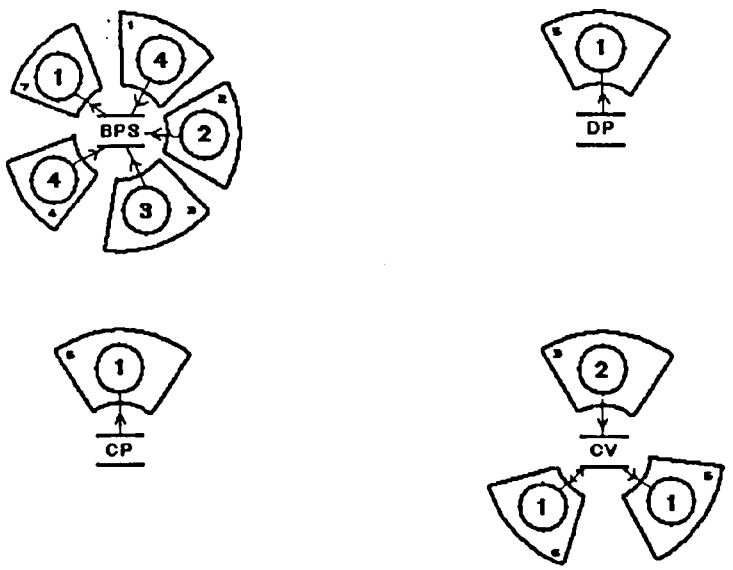

Figure 4.10: Data Dependency Rings for the Bottling Plant

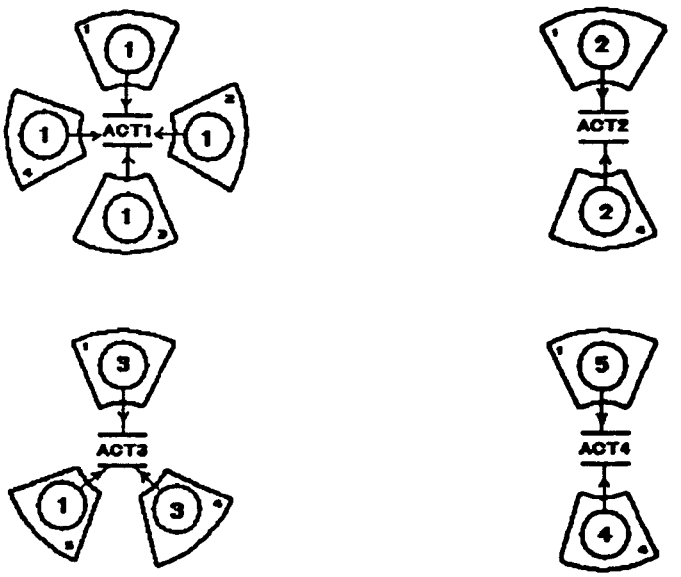

Figure 4.11: Data Dependency Rings for the Bottling Plant (cont.)

4.4 Uses of the Ring

The ring notation allows the designer to consider and express the implicit control flow between concurrent tasks and transactions necessary to maintain the consistency of shared data entities. 'Dense' rings i.e. those with many tasks and transactions listed around the edge represent a potential resource bottleneck of the associated data entity. These rings show that data entities, if possible, should be split into sub-entities. The ring notation also expresses, if perhaps a little clumsily, any enforced control flow that the designer imposes on the execution of some of the transactions i.e. the application requirements constraints. These are shown as the 'waits-for' markings and selection lines at the edge of the ring.

Besides these uses, the information contained in the ring notation can be applied to the automatic generation of the transaction precedence graphs for each task. The rings can also be used to determine the concurrency level for each data entity i.e. the number of useful copies of each entity in addition to guiding the allocation of data/tasks to processors in an implementation. These uses will now be described.

4.4.1 As a specification of the Database Usage

The ring notation does have some uses in expressing the role of each database entity in the real-time system. The ring notation shows us the following information :

- The tasks that use each data entity
- The transactions that use each data entity
- The entities that are going to be heavily used and that represent potential resource bottlenecks in the system
- The relationships between the different entities. If, for example, a transaction reads one entity and writes to another entity, then there is an implicit relationship between the two entities. This relationship is seen by a common transaction in the DDRs for each of the related entities. The ring notation makes this relationship clearer than for example having to follow data flow between entities in a dataflow diagram.

4.4.2 Generating Transaction Precedence Graphs

When the data dependency rings for each data entity have been drawn, we can generate the transaction precedence graphs. The secondary data dependency rings contain the partial orderings information necessary to meet the requirements of the application. The rings also embody the selection and iteration information. To generate a complete transaction precedence graph for each task, we need to consider the remaining control flow necessary. This control flow is necessary to prevent conflicting transactions from executing concurrently.

The data dependency rings show where any two transactions within a given task conflict. If two transactions appear in the same data dependency ring and either of the transactions is a write to the entity, then the transactions are said to conflict and some control is necessary between them. This control will manifest itself as an arc (i.e. control line) between the conflicting transactions in the transaction precedence graph. If two transactions do not appear in the same ring or neither of the transactions is a write then there is no conflict and the transactions are allowed to execute concurrently with no control flow between them.

As an example, consider figure 4.12. We can scan each transaction in each ring in turn and identify where control flow, or precedence

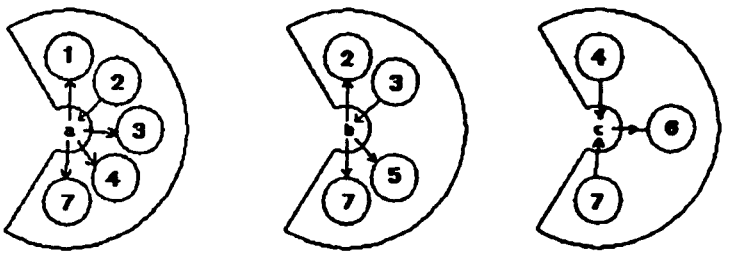

Figure 4.12: Example DDRs for a simple task

constraints are necessary to preserve the integrity of the database. A simple algorithm carries out this scan. The algorithm, shown in figure 4.13, also imposes the transaction orderings written on the outside of the ring (ARCs).

Applying the algorithm to the example in figure 4.12 we generate the following precedence constraints. Some of the precedences in figure 4.14 are redundant. For example, there is a precedence constraint imposed between transaction 3 and transaction 5 and there is a constraint imposed between transactions 5 and 7. It is not necessary to consider the constraint between 3 and 7. If transaction 5 waits for transaction 3 and transaction 7 waits for transaction 5 then since 'waits for' is reflexive, transaction 7 will implicitly wait for transaction 3. Any algorithm to generate precedence graphs should detect and remove these redundant constraints. The table in figure 4.14 represents the transaction precedence graph shown in figure 4.15. The order in which the precedences are applied can have an effect on the execution time for a task. For example, in figure 4.15, the necessary control flow between transactions 1 and 2 was shown as an arc between the two transactions. This means that transaction 2 must wait for transaction 1 to complete before it may start. If, however, we made transaction 1 wait for transaction 2 then we could get the better transaction precedence graph, shown in

```
For each Data Dependency Ring
  For each transaction in the ring and
  belonging to the same task

    compare with all other transactions
    in this ring

      if there is a data access conflict
      then impose a control flow (precedence
        constraint) between the two
        transactions

      if there is a explicit wait expressed
      then impose a control flow (precedence
        constraint) between the two
        transactions

      if the transaction selects between
      transactions
      then impose a selective control flow
        (precedence constraint) between the
        two transactions
```

Figure 4.13: Generating precedence graph from the transactions

Transaction	*Precedence constraints with*
1	2
2	1, 3, 4, 7
3	2, 5, 7
4	2, 6, 7
5	3, 7
6	4, 7
7	2, 4, 6

Figure 4.14: Precedence Constraints for the transactions in figure 4.12

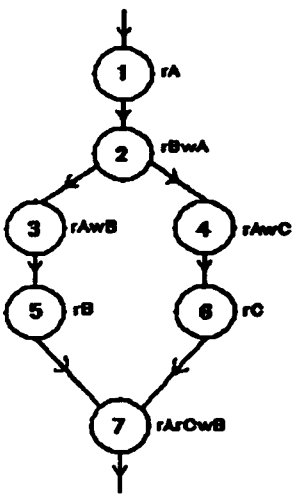

Figure 4.15: Transaction Precedence Graph

figure 4.16. This graph is better because there is more concurrency. Finding the best graph is a computationally intensive problem. For tasks with many transactions, it may be computationally infeasible to find the best graph.

There are heuristics to help in finding better transaction precedence graphs. One such heuristic is to group all those transactions that read a particular entity. These transactions can execute concurrently. Any transactions that write to the entity can then be serialised within the group. For example suppose we have the transactions:

1. read A
2. write A
3. read A
4. write A

In applying the above algorithm to generate the transaction precedence graphs we need to apply a precedence constraint between transactions 1 and 2. If we then considered transaction 2, we would apply a precedence constraint between this transaction and number 3. The final outcome would be a 'chain' of 4 transactions with no concurrency employed. A

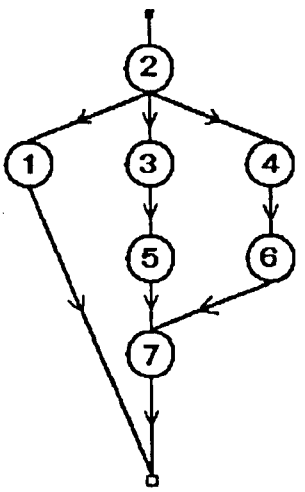

Figure 4.16: A better TPG

better solution would be to consider all the reads of A in one go and then serialise this with the writing transactions 2 and 4. This allows transactions 1 and 3 to execute concurrently.

4.4.3 Allocation Schemes

The information contained within the ring notation can be used to guide a data entity and transaction allocation scheme. The ring notation first of all shows the concurrency degree for the data entities. By grouping those transactions that read a data entity together we increase the 'concurrency level' for the transaction. The ring notation can thus be used to show the most useful number of copies of each data entity. As an example, consider the ring representation of a task shown in figure 4.17. It would appear that three copies of data entity 'A', two copies of data entity 'B' and three copies of data entity 'C' are required in order to allow the maximum concurrency for these three data entities. However, we cannot consider each ring in isolation when determining the concurrency degree. In figure 4.17, although in the ring for 'A', transactions three and four can execute concurrently, these transactions must be serialised due to a data dependency on the data entity 'B'. The concurrency degree for entity 'A' therefore goes down to two. The two

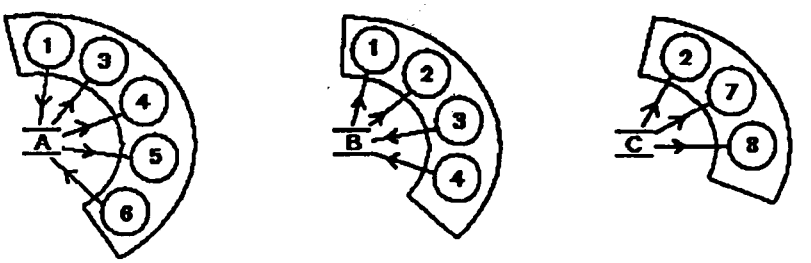

Figure 4.17: Example rings to demonstrate data entity concurrency degrees

copies of the entity will be used concurrently when transactions four and six are executing. The concurrency degree for data entity 'B' remains at two and the concurrency degree for data entity 'C' remains at three.

In an ideal implementation, enough processors are provided to meet the concurrency degrees of each of the data entities. In the example of figure 4.17 we have concurrency degrees of two (for entity 'B'), two (for entity 'A') and three (for entity 'C'). If each transaction in each data dependency ring were independent (i.e. it used only the data entity associated with the ring) then by summing the concurrency degrees of the entities we need seven processors to achieve the maximum concurrency for the task. This method of determining the number of processors required can often lead to wasted resource. The transaction precedence graph for this example shown in figure 4.18 has a maximum 'width' of four. This is the optimum number of processors that this task requires, given this transaction precedence graph. The reason for this is that many of the transactions in the example use more than one data entity. The width of a transaction precedence graph will always be less than or equal to the sum of the concurrency degrees of the data entities used by the task.

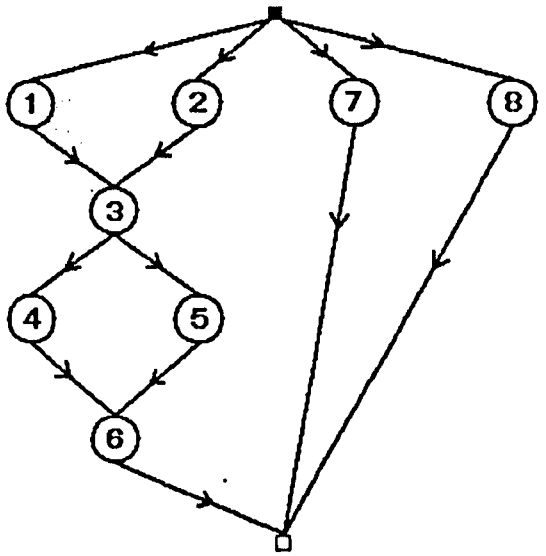

Figure 4.18: Transaction precedence graph for figure 4.13

After determining the width of a task and the concurrency degrees for each data entity, we can begin to allocate transactions and entites to the processors that will be used to implement the task. The allocation scheme proceeds according to the algorithm shown in figure 4.19.

Applying the allocation scheme to the example of figure 4.17 we get the allocation shown in figure 4.20. The allocation scheme tends to result in a allocation where the load on the processors is uneven. This can be seen in figure 4.20 where processor two has a greater allocation of transactions and data entities than processor 4. However, the scheme does ensure that concurrent transactions are placed on separate processors, and that all data entities that a transaction requires are stored on the same node as the transaction.

The allocation schemes so far is very naive. We have only considered allocating one task to a set of processors. We can generalise the scheme and, using the ring notation, determine the concurrency degrees for each entity given that more than one task exists. The allocation algorithm of figure 4.19 can then be used on each of the tasks in turn. A runtime mechanism must then be provided to ensure that if a task begins updating a shared entity and then must back off, the entity will be restored to a suitable state before a conflicting task has access to it.

The main weakness of this allocation scheme is that no consideration is

```
We will use W processors labelled 1..W
where W is the width of a task

For each transaction T(i) in the task

  If we can execute T(i) concurrently with
  T(i-1) (or i=1) then

      If there a processor with all the entities that
      T(i) requires, already allocated and such that
      this processor does not execute T(i-1)

          assign T(i) to this processor

      Otherwise

          find the processor in 1..W with the most of
          the required entities such that T(i) can still
          execute concurrently with other transactions
          on this processor. Allocate the remaining
          required entities and T(i) to the processor.

  Otherwise

      If the site of the previous transaction has all the
      required entities

          allocate T(i) to the same processor as the previous
          transaction.

      Otherwise

          If there are copies of the required entities still
          to be allocated

            allocate the required entities and T(i) to the same
            processor as the previous transaction.

          Otherwise

            allocate T(i) to a processor that has all the
            required entities
```

Figure 4.19: Allocating Entities and transactions to processors

Processor	Data Entities	Transactions
1	A B	1 3 4
2	A B C	2 5 6
3	C	7
4	C	8

Figure 4.20: Allocation of figure 4.13 to four processors

given to the temporal properties of the tasks. Providing suitable run-time mechanisms to ensure the correctness of shared data entities after preemption of tasks is discussed in the next chapter. In addition, evaluating the allocation scheme for the temporal correctness of the tasks is also considered. Should the allocation scheme result in a system which fails to meet the deadlines of all those time critical tasks, iteration of the design process is required. This may involve redesigning individual transactions in order to reduce their execution times or redesigning a complete task, again to reduce the execution time.

An initial first pass allocation would procede by applying the algorithm in 4.19 to each of the tasks in the real-time system. This would generate the best possible allocation for the concurrency expressed in the TPGs for the tasks (ignoring the side effects of update propagation to multiple copies of data and increased communications between multiple processors). For any non-trivial real-time system, this would result in a allocation that required hundreds of processors to implement. We need some way of reducing the number of processors while at the same time ensuring that the temporal properties of the tasks are preserved.

Figure 4.21 describes a heuristic based 'reduction' algorithm. The aim of this algorithm is to combine the work of separate processors in order to reduce the number of processors used in an implementation. The algorithm should primarily be used to group the work of the transactions within a task (transaction reduction). The algorithm can then be used to group the tasks together onto combined processors (task reduction). The reduction heuristic aims to even the load across a limited set of processors. Since the algorithm is based on a heuristic, it does not always produce the best allocation of tasks and transactions to processors. The alternative would be to generate the search space of every conceivable allocation of transaction to processor and then examine this space to find the best. This approach is not suitable for any but the most trivial of allocations.

In grouping transactions together, it is possible that the processor can-

```
The load on a processor is defined as the total execution time
of all those transactions 'sited' at that node.

Repeat

  combine the transactions and data entities of the two
  processors with the smallest loads into a single processor.

until the number of processors used is less than the
maximum number allowed in the implementation.
```

Figure 4.21: Reducing the number of processors in an implementation

not execute all the transactions before the deadline of a task. The following chapter describes how to check the feasibility of an allocation. Should an allocation not be feasible then the tasks could be split up and a different allocation tried.

An alternative grouping method is based on assigning a transaction to the processor which has the best subset of the entities that the transaction requires. This method is as follows. For each task there exists a 'cluster' of 'logical processors' available to execute the task. Within each task, there are enough logical processors to fullfill the maximum concurrency available within the task. For example, the cluster for the task of figure 4.18 has four logical processors; the transactions of the task are allocated such that concurrent transactions are on separate logical processors and transactions that must be serialised are, where possible, placed on the same logical processor. Given the clusters for a large set of tasks, we are bound to have more logical processors in the allocation than available physical processors for an implementation. We now need to group the actions of logical processors together in order to allocate the work to the physical processors. Rules of thumb, or heuristics, exist to help in this grouping. The first is that we shouldn't assign the work of logical processors within the same cluster to the same physical processor where possible. This maintains the degree of concurrency within the task where permitting. The second rule of thumb is to place a logical processor to a physical processor that already has allocated to it some (largest) subset of the entities that the logical processor requires. This ensures that the logical processors are placed on the physical processors with the best subset of data entities. An example of this extended allocation scheme in described in Appendix B, for the Ship Control System.

4.4.4 Critical regions and generating back off information

The previous chapter described the concept of a critical region and showed how a task may be considered as a collection of connected critical regions, or a molecule. In this section we will show how the information expressed in the DDR notation can be used to generate the back-off information needed when critical regions are interrupted.

Critical regions on different data entities are connected through shared transactions. A transaction may read multiple data entities or it may read a set of data entities and write to one other data entity. In these cases, the critical regions are connected and we must determine the effects on a critical region given that the connected critical region is being backed off. A critical region on an entity is backed off if some other more 'urgent' task needs to access the entity. The critical region is known as a primary critical region and the backoff is known as a primary backoff. We need to determine the primary backoff point for this region. This backoff point is at the start of the critical region and may be found through examination of the transaction precedence graph for the task. The backoff point is the first use of the data entity.

We now need to consider the back off points for any connected critical regions. A connected critical region is one in which the data entity either directly depends on the primary data entity or is updated during the lifetime of the primary critical region. In the first case, a transaction will read from the primary entity and write to the dependent entity. This will appear in the ring notation as a transaction common to the two rings and where one instance of the transaction is a writing transaction. In the second case where the dependent critical region is updated during the lifetime of the primary critical region being backed off but where there is no direct dependency between the two data entities. This would occur where ARCs are introduced into the task.

Where a primary critical region is to be backed off, we need to consider all connected critical regions. The values of data entities for all connected critical regions should be restored to their state on entry to the primary critical region. In this way, several problems will be avoided and the task will always see a consistent database state even through interruption by other tasks. The first problem that is avoided is where an independent data entity is updated more than once. Suppose the following represents part of a task where an update to data entity 'B' occurs within the critical region of data entity 'A'. The transactions appear in the transaction precedence graph in the sequence specified

by their number.

```
1. rB
2. wB
3. rA
4. rA
5. wB
6. rA
7. rA
```

Suppose the task is interrupted between transactions 6 and 7 by another task which changes the value of data entity 'A'. To ensure that on restarting the above task, consistent values of 'A' are read, we need to back off the task to the start of the critical region. The task on restarting will begin execution from transaction 3. This means that transaction 5 is executed a second time and the value of 'B' may become 'corrupt'. To prevent this problem, we need to restore the value of 'B' to its state on entry to the primary critical region that is being back off i.e. that region for data entity 'A'.

The second problem occurs when there is a dependent data entity updated within the critical region being backed off. Consider the following task.

```
1. rB
2. rA
4. rB
3. rAwB
4. rA
5. rA
```

Suppose the task is interrupted between transactions 4 and 5 by another task which updates data entity 'A'. To ensure consistency in the reading of 'A', we have to back off the critical region to its start before restarting the interrupted task. Suppose this is all we do, then on restarting the task, the value of 'B' read in transaction 4 will be an updated value created during the previous, aborted, execution of the task. Data entity 'B' therefore has no integrity. To prevent this, we should restore the value of 'B' to its state on entry to the critical region for 'A'.

The rule then is: where any data entity is updated during a critical region its initial state on entry to that critical region should be restored if that critical region is restarted. The dependent data entities can be identified through examination of the ring notation and transaction

precedence graphs. A list of dependent data entities can be constructed for each critical region. The state of these can be saved on entry to the critical region. This state can be thrown away on completion of the critical region. If there is too much state to be saved, the alternative is to save the state of each entity as its critical region is entered. Where we need to back off to the start of a critical region, then we should also back off to the start of any connected critical region. This was described in the previous chapter. The further the task is backed off, the more work has to be repeated. There is a trade of between how much state is saved and how much work needs to be repeated.

4.5 Summary of the Method

The real-time design methodology so far comprises the following steps:-

1. Identification of real-time triggers.
2. Decomposition of system into subsystems (grouping of related triggers).
3. Decomposition of subsystems into tasks.
4. Preliminary design of real-time database.
5. Decomposition of tasks into transactions.

Following these steps, we can now proceed with

- Construction of Data Dependency Rings as a diagrammatic aid.
- Automatic generation of Transaction Precedence Graphs.
- Determine useful concurrency degrees (number of copies) of each data entity.
- Automatic generation of a transaction/data entity allocation scheme.
- Automatic generation of back off information.

Chapter 5

A Run-time Environment

Saturn, the Bringer of Old Age

5.1 Introduction

The previous chapter described a simple methodology to transform an informal description of a real-time system into a design for an implementation. This design, although meeting the functional requirements of the application fails to address the, perhaps equally important, non-functional requirements. Such non functional requirements for a real-time system include the ability to meet the real-time deadlines of the tasks.

This chapter is divided into three sections. In the first two sections we consider the principles and techniques for meeting hard real-time deadlines. In the third section we describe a run time environment that is based on the model described in Chapter 3. This run-time environment ensures that both the functional requirements of the application and the real-time deadlines of the tasks are met.

5.2 Real-Time Scheduling - A Survey

This section considers the problem of real-time scheduling: it describes the problem of deterministic real-time scheduling; introduces a taxonomy of real-time scheduling methods and then describes examples of scheduling mechanisms in each part of the taxonomy.

5.2.1 Deterministic Scheduling

In a *soft* real-time system, the tasks are performed by the system as fast as possible; the tasks are not constrained to finish by specific times [SCS88]. Metrics for the evaluation of soft real-time systems include average response times for tasks and system throughput. In the alternative, hard real-time systems, the tasks have to be performed not only correctly but according to strict temporal constraints. That is to say that each task has a deadline; if the task does not complete its work before this deadline then the system is deemed to have failed. The design methodology presented in this work is intended for applications with a high predominance of hard real-time tasks. Consequently, this chapter is primarily concerned with hard real-time scheduling. We do however present an overview of scheduling of soft real-time tasks.

5.2.2 Nature of Tasks

Chapter 3 introduced the temporal characteristics of tasks in a real-time system. The important temporal characteristics of the tasks are

1. The *arrival time.* This is the time that the task is triggered.
2. The *ready time.* This is the earliest time that the task can begin execution. The ready time is always greater than, or equal to, the arrival time.
3. The *worst case execution time.* The execution time of the task is always less than or equal to the worst case execution time. In our work, the worst case execution time is evaluated by analysis of the transaction precedence diagram for a task. The 'longest path' through the TPG represents the worst case execution time.
4. The *deadline.* The latest time by which the task must have completed its execution.
5. The *minimum repeat time.* This is the earliest time after a task triggering that it may be re-triggered. For periodic tasks, the minimum repeat time is the period of the task.

In a static real-time system, all the above timing information is known before run-time. For a dynamic real-time system, not all this information is known before run-time. Many real-time scheduling techniques draw a distinction between periodic and non-periodic tasks. A non-periodic task has arbitrary arrival times and deadlines. A periodic task

is one that is executed exactly once per period P. A system constructed from periodic tasks, is ammenable to static analysis. The scheduling analysis described later in this chapter converts a set of non-periodic tasks into a periodic set that represents the absolute worst case of task triggerings. Static analysis can then be performed on this task set.

The Guarantee Ratio and Optimal Algorithms

A schedule is defined to be the order of execution of the tasks and the start times for execution of each task. The function of a scheduling algorithm is to determine whether a schedule for executing a set of tasks exists such that all their constraints are satisfied and to generate this schedule. A static algorithm determines the schedule off-line using the timing information of the tasks. A dynamic scheduler determines the schedule on-line and progressively as more information about the tasks is known.

A scheduling algorithm is said to guarantee a new task if a new schedule can be found such that the timing and resource constraints of all existing tasks as well as the new task are satisfied. A static algorithm aims to guarantee all tasks. A measure of the performance of a dynamic scheduling algorithm is the number of tasks guaranteed versus the number of tasks that arrive. This is known as the guarantee ratio.

For static scheduling, an algorithm is optimal if it always produces a schedule with guarantee ratio of 1 whenever any other algorithm can do so. A dynamic scheduler is optimal if it produces a schedule with guarantee ratio of 1 whenever there exists a static algorithm that can do the same given the timing constraints. [SCS88] states that finding an optimal dynamic scheduling algorithm is difficult and computationally intractable. [SCS88] also states that an approximate algorithm with the highest guarantee ratio is considered to be better than one with a low ratio. However, unless the guarantee ratio is 1, there exists a scenario of task triggerings such that the dynamic scheduling algorithm cannot guarantee all tasks. We propose that to achieve hard real-time scheduling some static analysis of the task set is necessary. A method to carry out this analysis is described after a survey of relevant work.

5.2.3 Taxonomy of the scheduling solutions

A real-time scheduling algorithm may be described in terms of various characteristics. By combining these characteristics, different scheduling algorithms can be defined. The main characteristics are:

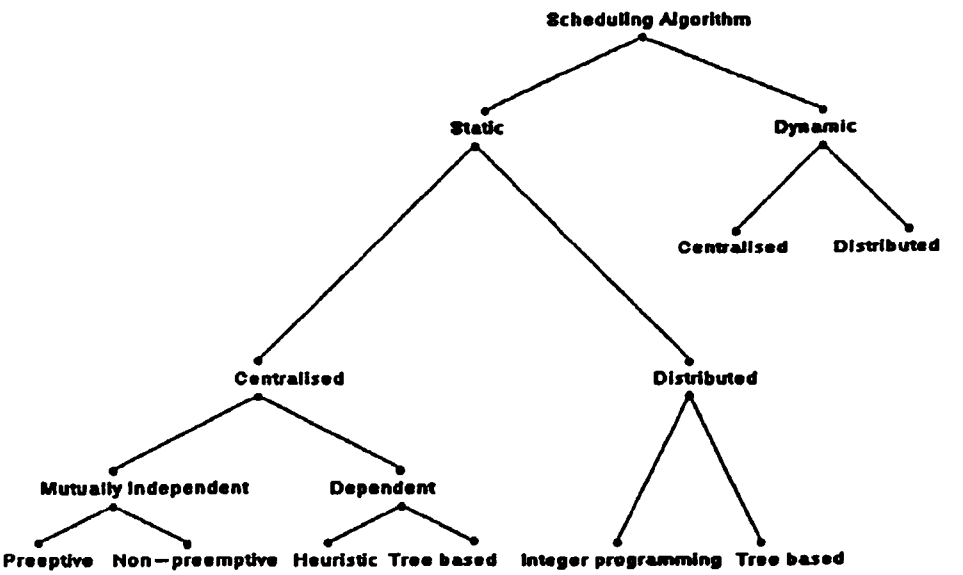

Figure 5.1: Taxonomy of scheduling solutions

- Static or Dynamic approach
- Centralised or distributed scheduling
- Preemptive or non-preemptive tasks

The taxonomy of solutions is shown in figure 5.1.

Static Approaches

A static scheduling algorithm can use information available at system build time or before, to determine a complete or partially complete, feasible schedule for the total set of tasks. The static scheduling algorithm is typically a load balancing scheme to assign tasks to processors.

Preemptive Approaches

If preemption is allowed among the tasks to be scheduled, then according to [SCS88], it is often possible to find polynomial-time optimal algorithms for statically scheduling the tasks. Static scheduling methods typically use a simple scheduling scheme at run-time to decide which task from a set of triggered tasks, should be executed next. A static analysis determines the effectiveness of the algorithm.

For a set of periodic tasks, if a schedule can be found for the time interval between 0 and the least common multiple of all the task periods, then the same schedule can be correctly used for all repetitions of this same period. [Mar82] describes an earliest deadline first (EDF) scheduling policy for a set of periodic tasks on a multiprocessor system. The complexity of the scheduling analysis for this multiprocessor case is $O(m^2n^4+n^5)$ where m is the number of processors and n is the number of tasks. A performance analysis of this scheduling policy is found in [HHT89]. Other methods for scheduling a set of periodic tasks can be found in [Leu89]. Further work in preemptive scheduling can be found in [WKSG89].

[LL73] uses a similar approach to real-time scheduling. A simple scheduling policy is chosen and statically analysed for feasibility. This work defines 'overflow' of tasks to occur at a time t if t is the deadline for a periodic task that has not yet finished. A feasible scheduling algorithm is then defined as one which schedules the tasks such that no overflow can occur. [LL73] describes a simple algorithm for executing the real-time tasks at run-time. Each task is assigned a priority. A given task may be preempted by a task of higher priority. Each task has a critical instant. This is when the task is scheduled at the same time as a set of other tasks and such that all the other tasks have higher priorities. If the requests for all tasks at their critical instants are fulfilled before their respective deadlines, then the scheduling algorithm is feasible. As an example suppose we have two tasks with period T_1=2 and T_2=5 and execution time E_1=1 and E_2=1. Assume that task one has the higher priority. The critical instant for task two is when both task one and task two are triggered at the same time, t_0. In this case, task one can run from t_0 to t_1, t_2 to t_3 and t_4 to t_5. Task two can be run from t_1 to t_2. Both tasks meet all their deadlines; task one runs three times in the period of task two.

[LL73] suggests an inequality that must be satisfied for the schedule of a set of periodic tasks to be feasible. If T_1 and T_2 are the request periods of two tasks with execution times E_1 and E_2 such that $T_1 < T_2$ and L(X) is the lowest integer smaller than or equal to X, if task one is the higher priority task then following must be satisfied

$$L(T_1/T_2)*E_1 + E_2 <= T_2$$

If task two were of the higher priority then

$$E_1 + E_2 <= T_1$$

Intuitively, the first relationship states that task one is executed $L(T_1/T_2)$

times and provided this many executions of task one and a single execution of task two can all be executed within one period of task two then the task schedule is feasible. The second relationship states that if task one and two can both be executed within the smaller period with task two executed first then the task schedule is feasible. [LL73] states that a reasonable rule for assigning priorities is to assign the higher priority to those tasks with the faster triggering rate. This is known as a rate monotonic priority assignment and leads to an optimal algorithm.

[Mar82] describes an alternative method of determining whether a feasible schedule for a set of tasks exists. Unlike the work of [LL73], this method can be applied to task sets that are not periodic. The method needs to know the start (trigger) times and deadlines of each task. In addition the processing requirements for each task should be known. With N tasks, a time-line can be divided by the trigger and deadline times of the tasks to yield 2N+1 regions. If t_i is the i^{th} smallest value among the trigger and deadline times then the ith interval is the period from t_i to t_{i+1}. A task is available for processing in the ith interval if its trigger time is before (or at) the start of the interval and its deadline is after (or at) the end of the interval. Within the interval, the set of available tasks does not change. Given the amount of processing to be done on each task within the interval, scheduling these tasks within the interval is an instance of the problem where all tasks have the same trigger and deadline times. [Mar82] states that it is sufficient to find the amount of processing to be done on each task within each interval and then use an algorithm such as that in [GS78] to schedule the tasks.

The main problem with the work in [Mar82] is that the tasks are assumed to have known trigger times. There is no consideration is made of tasks that repeatedly trigger, whether periodic or not. Perhaps repeated triggerings of an individual task could be considered as independent instances of a task and consequently treated like different tasks. Other work on preemptive scheduling in multprocessor systems can be found in [MC70]

Non-preemptive Approaches

Non-preemptive scheduling is more difficult than preemptive scheduling; many non-preemptive scheduling problems have been shown to be NP-hard [SCS88]. However, much work has been done on restricted problems. For uniprocessor systems, it can be shown that the earliest deadline first algorithm is optimal for scheduling a set of non-preemptable tasks with the same trigger time. A similar restricted problem is discussed in [JBW86]. This work imposes additional severe restrictions on the task set. It is assumed that each task can be com-

puted in unit time but that a given task may use an arbitrary number of processors. The work describes an algorithm to schedule a set of one processor tasks with a set of k processor tasks where k is less than the maximum number of processors available. The algorithm states that the k processor tasks should be scheduled first. The one processor tasks are executed on the idle processors during the execution of the k processor tasks and also when the k processor tasks have completed. The problem of scheduling a set of tasks that require 1,2,3..k processors is then seen as being a simple linear programming problem. To apply this work to a task set where the tasks have different execution times, the tasks could be decomposed into a set of subtasks with the same execution time. The hierarchical nature of the task as described in Chapter 3 aids this decomposition. The methods proposed in [JBW86] could then be used to determine a schedule for these subtasks.

Mutually Dependent Tasks

The techniques of [Mar82], [LL73], [Sah76], [GS78] and [JBW86] are intended for tasks sets where the tasks are independent with respect to data use. In reality, tasks are often mutually dependent. As described in the previous chapter, tasks often share data and this necessitates a serialisation of critical regions. For the mutually dependent scheduling case, a correct schedule satisfies the temporal constraints of the tasks as well the task ordering requirements. Preliminary work has shown that scheduling non-preemptable tasks with deadlines and arbitrary precedence constraints can be solved with the earliest deadline first algorithm in $O(n^2)$ time. In addition it has been shown that a preemptive schedule for mutually dependent tasks exists if and only if a non-preemptive schedule exists [SCS88].

There are two methods of scheduling mutually dependent tasks: heuristic approaches and those based on searching a tree based precedence graph. [KN84] describes a heuristic approach. An optimal schedule to large scale problems can be found out within a time limit that grows exponentially with the size of the task set; the method is not suitable for dynamic scheduling of tasks. The method is only really suitable for task sets where all tasks are triggered at known times.

The method proceeds by constructing a task precedence graph. This is similar to the transaction precedence graphs of Chapters 3 and 4. Each node represents a task and an arc represents a constraint that is necessary between the tasks. This constraint enforces serialisation of conflicting data accesses. Each node is augmented with a weight representing the processing cost of the associated task. The graph has a single root (entry) node and an exit node. These represent the first and

last tasks. Each node is assigned a depth based on the maximum value of the processing weights from the entry node to the node in question. The tasks are sorted based on this depth to generate a priority queue. If the depth of two nodes is the same, the method provides a heuristic that the node with the greatest number of arcs should be sorted first. This heuristic is based on having the task with the greatest number of dependents begin execution first. Using a set of K processors, assign the first N tasks to the K processors such that each of the N tasks is not waiting for some other task to complete and N<K. When a processor becomes free, schedule the next task in the priority queue provided it does not wait for an, as yet, uncompleted task.

[KN84] claims that 67% of the schedules generated by this method are optimal and in addition 87% of the non-optimal schedules are within 5% of the optimal solution. The method has many disadvantages. The most significant of these is that it is only valid for task sets where all tasks are active at the same time. There is no consideration of arbitrary sets of tasks being triggered. We could apply the technique to determining schedules for all periodic tasks in a system and then use some other technique for determining the schedule for non-periodic tasks. Applying the method to the model of Chapter 3, it would be inappropriate to apply a constraint between two tasks based on the conflicting access to one entity. In reality constraints would be imposed between the conflicting critical regions of the tasks. The critical regions would then be the nodes in the graph analysed by the method of [KN84].

The alternative to the heuristic approach is to use a tree based technique. [MC70] describes such an approach. The tasks to be scheduled are inserted into a tree with the tasks that should be scheduled first at the leaves of the tree and the last task to be executed is placed at the root. The algorithm for generating the schedule proceeds by allocating a processor to each of the tasks at the leaves. These tasks are now executed. Each time one of the following situations occurs a processor is re-assigned to meet as many of the deadlines as possible.

The two situations are as follows:

1. When a task has completed execution

2. On reaching a point such that, that if we continued the present assignment, some tasks would be computed at a faster rate than other tasks that are further from the root (i.e. those with a higher priority). This is determined by considering the height of the node in the same way as determining the depth of a node in the method of [KN84]

As with [KN84], the main problem with the method in [MC70] is that all tasks are assumed to be active at the same time. This is inappropriate for a system where the tasks are independently triggered and where it is unknown before run-time what tasks are to be scheduled.

Further work on scheduling tasks in a centralised environment can be found in [Man67], [Moo68].

Distributed Allocation Approaches

Even without the addition of timing and precedence constraints among tasks, the generation of schedules for a set of tasks is known to be difficult. Finding schedules for distributed systems is even more complicated. Finding the optimal schedule for a set of tasks and three processors is known to be NP-hard [Bok81]. The literature contains two methods for allocating tasks to processors in a distributed system to generate real-time schedules. The first method is a tree based method; the second method is an *integer programming* method.

[Bok81] describes a tree based method for the allocation problem. The method aims to minimize the sum of the execution times on each processor in a distributed network as well as minimizing the communication costs. Using the model of Chapter 3, communication between tasks is not a functional requirement, it is only required to synchronise updates to shared data entities. The method described in [Bok81] does not guarantee the deadline of a particular task. Instead, the algorithm ensures that each task gets the best possible chance at completing its work before the deadline. The method is to construct two 'cost' matrices. The first gives the cost of executing each task on each of the processors. The second gives the cost of communication between each pair of tasks. An assignment graph (tree) is then constructed. This contains a node for every combination of processes on processors. Every communication path between these process allocations is then added to the tree, and the tree augmented with the communication costs. A shortest tree algorithm is then used to find the optimal path through the graph. One recognised disadvantage of this graph theoretic approach to finding an optimal allocation is that the time complexity of the algorithm is high when the number of processors is more than two [ELT82]. Again, the main disadvantage of this method is that it assumes that all tasks are active at the same time. The work has been extended to the case where there is a probability of executing one task directly after another. This is described in [Tow86]. The work of [Bok81] and [Tow86] can be extended to determine schedule length from the allocation pattern; on their own these methods do not consider the real-time deadlines of the tasks.

The second method to allocate tasks to processors in a distributed system is based on the 'integer programming' method [ELT82], [WCE80]. [ELT82] describes a technique for generating an optimal solution to the allocation problem when faced with a set of tasks that must be executed within a certain 'port-to-port' time. Like the graph theoretic approaches of [Bok81], [ELT82] uses information such as the communication cost between any two tasks, the inter-processor distance between two processors and the cost of executing a given task on a given processor. As in [ELT82], an allocation graph is constructed, but a 'branch-and-bound' technique restricts the number of nodes in the graph. This branch and bound method uses nine rules to restrict the number of nodes. These rules select the best node from the current set of nodes. The main problem with the integer programming techniques is its exponential computational cost. Other scheduling work involving branch and bound techniques can be found in [Ma84].

In addition to the task allocation problem, some other useful research has been carried out to study the effects of various schedules for a set of distributed real-time tasks. One of the most important tasks that a real-time system can do is to determine whether a given schedule meets the deadlines of all the tasks. [Lei80], [LY86] provides a way of doing this. An algorithm is provided that is used to determine an upper bound to the execution time for each task. It is easy to determine the execution time of a task in isolation. To determine the upper bound on the execution time however, it is necessary to determine how long a task is blocked by the actions of other tasks. This blocking is because other tasks hold resources required by the task.

Determining the worst case blockage is a computationally intractable problem for a large set of tasks. As a result [LY86] adopts a slightly different strategy. Instead of finding the worst case blockage, the algorithm presented starts with every task simultaneously blocking every component that it could possibly block. From this set of blockages, the 'impossible' ones are removed and the resultant set of blocks transformed into the worst case blockage. As an example of an impossible block, suppose we have sequential tasks A,B,C and D executing in parallel with the sequential tasks X,Y and Z. Assume X blocks A,B and C and Y blocks A. If X were really blocking C then Y could not block A, since for Y to block A, X must have run which would imply that X is not blocking C. This type of incompatibility between blocks is removed from the consideration set and replaced by one combined block that reflects the maximum effect of the constituent incompatible blocks.

In addition to the incompatible block transformation [LY86] provides

several other transformations on the set of blockages. Each transformation reduces the set of total blockages that need to be considered. The work provides a method for determining a reasonable estimate of the upper bound of the worst case execution time for a task set. This technique can be used with other scheduling methods to determine the effectiveness of a schedule before execution time.

Dynamic Approaches

The objective of dynamic scheduling is to find the optimal schedule given that the trigger times of the real-time tasks are not known in advance of system execution. All the static scheduling techniques require some knowledge of the trigger times prior to scheduling. This section briefly describes some of the approaches that have been taken to solve the dynamic scheduling problem. There are two categories of dynamic scheduling. In the centralised approaches, one node in the computer system is responsible for making the scheduling decisions. In the distributed approaches, scheduling decisions are made concurrently at more than one site in the distributed network of processors.

Centralised Approaches

In theory, any static scheduling algorithm that generates an optimal schedule can be used on-line to guarantee tasks dynamically. However, most of the algorithms, such as those discussed in the previous section, are optimal for static scheduling but not for dynamic scheduling because of their complexity. [SCS88] states that for multiprocessor systems, there can be no optimal algorithm for scheduling preemptable real-time tasks where the arrival times are not known before hand. As a result, heuristic algorithms become more significant in the problem of scheduling real-time tasks dynamically. Although, without the start times, we cannot have an optimal solution, there are ways of testing if a heuristic generates a correct (but not necessarily optimal) solution.

Distributed Approaches

The main problem that must be addressed by distributed dynamic scheduling policies besides generating the schedule quickly enough, is stale data. In a distributed system, tasks may be dynamically triggered at any node in the system. The status of each node is constantly changing and cannot be predicted with any great certainty beforehand. A distributed scheduling algorithm that attempts to generate a schedule that is for the good of all nodes in the system requires state information from some subset, if not all of the other nodes. This information can only be acquired at run-time, and because of communication delays,

no propagated state information is guaranteed to be up-to-date. The scheduling algorithm often has to make its decisions based on stale information. Often, distributed, dynamic schedulers consist of two parts; a local scheduler that schedules those tasks that can be executed locally, and a distributed scheduler that schedules those tasks that either need remote data or are to be executed on a remote node.

Few really successful dynamic real-time schedulers have been reported in the literature. There is however, much work on the related problem of dynamic load balancing; this goes part of the way to solving the problem. By far the most significant research into dynamic, real-time load balancing is reported in [WZS87], [RS84] and [JSC85], preliminary research papers from the development of the real-time 'Spring Kernel'. The Spring Kernel is a real-time operating system that advocates a new paradigm based on the notion of predictability and guaranteed deadlines [SR89].

[RS84] and [JSC85] present a scheduling algorithm that works dynamically on loosely coupled distributed systems for tasks with hard real-time constraints. The research presents a model of a hard real-time system consisting of a set of nodes. Each node has a set of periodic tasks. At system initialisation time, it is verified that the given allocation of periodic tasks allows each task to satisfy its deadline constraints. Any of the previously described techniques for static scheduling of periodic tasks in a distributed system can be employed. In addition to the periodic tasks, non-periodic tasks may be triggered at any node at any time. These sporadic tasks must be scheduled dynamically given that their CPU requirements and deadlines are known in advance. The aim of the scheduler is to guarantee all periodic tasks and as many of the non-periodic tasks as possible using the resources of the entire network.

Each node in the system consists of two parts; a local scheduler and a distributed scheduler. The local scheduler is responsible for ensuring that all local periodic tasks are guaranteed. This is done by maintaining a list of all periodic tasks and executing them according to the earliest deadline first heuristic. Should a non-periodic task be triggered at a node then, if the node has enough surplus processing power between when the task arrives and its deadline, the non-periodic task can be guaranteed at that node. If this is the case then the task is added to the guaranteeable task list. The surplus processing power available at a node is worked out by considering the deadlines and last possible start times for all the guaranteed tasks. The local scheduler itself is a cause for concern. If the local scheduler can preempt any currently executing task to determine what to do with a newly triggered non-periodic

task, then there is a danger that the preempted task will not meet its deadline. If however, the scheduler waits until the completion of the current task, then the newly triggered task may miss its deadline. To get around this problem [RS84], [JSC85] suggest that a local scheduler should be a periodic task. The period of the local scheduler determines the number of non-periodic tasks that are guaranteed.

What happens if the newly triggered non-periodic task cannot be scheduled on the current processor i.e. there is not enough surplus to guarantee the task. There are two approaches that the local scheduler can take to guarantee the task on another node. These approaches have been named 'bidding' and 'focussed addressing' [JSC85]. In focussed addressing, the node at which the new task has been triggered considers the surplus of all other nodes. If a particular node has surplus significantly greater than the computation time of the new task then this seems like a good candidate for executing the new task. The task is then sent to this node where it is treated like a newly triggered task. If the surplus at this node has changed since the original determined that this node was suitable, then the task may be forwarded a second time. In bidding, the original node checks the surplus of all other nodes. The task is then sent to the node with the greatest surplus. The difference between bidding and focussed addressing is that in bidding a node sends a request message to each node to find the surplus at that particular time. In focussed addressing the surplus is piggybacked onto other communications and is stored in a table at each node. Focussed addressing can use out of date surplus information whereas bidding requires a greater communications overhead.

[JSC85] presents some performance figures for the dynamic scheduling scheme used by the Spring Kernel. By tuning the periods of the scheduling functions and adjusting other system parameters, a guarantee ratio of 98% has been reached. However, the remaining 2% of tasks that fail to meet their deadlines is cause for concern. By definition of a hard real-time system, if any task is not guaranteed then the whole system is deemed to have failed. With the dynamic scheduling method used in the Spring Kernel, there is always a possibility that a task might fail.

5.3 A Combined Static/Dynamic Approach

5.3.1 Introduction

The function of a scheduling algorithm is to determine, for a given set of tasks, whether a schedule for executing the tasks exists, such that the timing, precedence, and resource constraints of the tasks are satisfied. If such a schedule exists, then the task set is 'sound'. In addition, the scheduling algorithm should generate the schedule for a given scenario, if one exists [SCS88]. In a hard real-time system we must guarantee that all tasks meet their deadlines and resource/precedence constraints. If any task does not meet its deadline then the system fails.

Generating optimal schedules before run-time is not only a difficult problem but it also relies on knowing the trigger times for the tasks. For real-time systems that consist of periodic tasks only, the analysis can be carried out. For systems that have a combination of periodic and non-periodic tasks, conventional static analysis to generate a schedule is harder. For these systems, dynamic scheduling techniques such as those described in [JSC85] rely on a run-time scheduling heuristic to decide where to execute a newly triggered non-periodic task. Often the load on the system at a particular instant cannot be determined and so there is a real danger that a particular hard real-time task will fail to meet its deadline.

This section describes a static approach to the analysis of a dynamic run-time heuristic used to schedule a set of tasks. The static analysis doesn't generate a schedule as in other static scheduling techniques; the analysis simply determines the effectiveness of the dynamic heuristic. If the analysis shows that the heuristic is successful, at run-time the heuristic will correctly decide what to do with newly triggered tasks such that all deadlines are met. This approach is similar to that used by [LL73].

Some information about the task set is required before the static analysis can be carried out. For each task, we need to known the worst case execution time, the deadline (relative to triggering time) and the minimum re-trigger time (MRT), again relative to the previous triggering time. The worst case execution time can be found by analysis of the transaction precedence graph for a task. The individual execution times, of all transactions on the longest path through the graph, give an indication of the worst case execution time. The minimum re-triggering time may be at the deadline, or after the deadline, of the current invocation of the task. The MRT may alternatively be before the deadline

of the current task invocation. The second case represents overlapped triggerings of the tasks.

The static analysis is carried out by imposing a periodic model of task triggerings on the otherwise aperiodic, asynchronous, task set. A static analysis can be carried out on this periodic model. For a given set of aperiodic tasks, we assume that each task is triggered at a time T_0 and is continually re-triggered at its MRT. This is considered to be the absolute worst case that the scheduling mechanism will have to deal with. The static analysis proceeds by considering the effectiveness of the scheduling heuristic when faced with this worst case scenario. By considering this case, if the scheduling policy is successful, then it is also considered to be successful in all other cases i.e. when tasks are not necessarily re-triggered at their MRTs.

5.3.2 Chosing a Scheduling Heuristic

The run-time scheduling heuristic chosen for analysis is the Earliest Deadline First (EDF) selection policy. This scheduling policy is used several times in the literature [Mar82]. In the EDF policy, if there is a choice to be made between two 'conflicting' tasks then the task with the soonest deadline is chosen to be executed next. Tasks are said to conflict if they require conflicting access to any shared resource (processor or data entity). There are various extensions that need to be considered with this policy. For example, if a processor is busy on a task when a new task triggers with a sooner deadline than that in execution, the scheduling mechanism has two choices: the first is to continue to the end of the current task and then start the new task; the second choice is to preempt (interrupt) the current task and start the new task.

The Earliest Deadline First scheduling policy was chosen because of its already widespread use in real-time systems. The policy is also simple to implement; this implies that the overhead imposed on the system because of scheduling decisions is minimised. Other simple scheduling heuristics could have been considered. Among these are:

- *Least Slack First.* The slack is the time between completing a task and the deadline of the task.

- *Greatest Utilisation first.* The utilisation of a task is the ratio of the trigger time subtracted from the deadline and the worst case execution time.

Each of these scheduling policies is simple to execute. The policies can also be localised. Each node in a system has a set of tasks that it can execute. Each node has a local scheduler based on simple scheduling policies. No inter-node communication is required to make a scheduling decision. For this work, EDF was chosen.

5.3.3 Worst Case Scenario Analysis

In analysing a scheduling policy we must determine the worst case scenario that the policy has to deal with. If the policy works in this worst case then it is assumed to work in all other cases that it is presented with. Given a scheduling policy such as EDF there are many tests that can be carried out on a task set to determine its effectiveness. This section describes such a set of tests.

SubTasks

In determining the effectiveness of a given scheduling policy, we cannot treat the task as an indivisible scheduling unit. Instead, the task is broken up into subtasks that are allocated to separate processors. The task is also considered as a set of critical regions each associated with a given entity. In the scheduling analysis that is to follow where we refer to a task, we are referring to that subtask of the real-time task that is allocated to a particular processor. In considering the timing characteristics of this subtask, we cannot use the characteristics of the parent real-time task. Instead we must construct the timing characteristics of the subtask. For example, suppose we have a task that consists of three sequential transactions T1 (exectime of 5 TUs), T2 (exectime of 10 TUs) and T3 (exectime of 5 TUs). Transactions T1 and T3 are placed on processor P1 and transaction T2 is on processor P2. Suppose the task has a deadline of 50 TUs and MRT of 60 TUs.

In considering the subtask on P2, when this is triggered, we have already completed T1, the deadline for this subtask is thus 5 TU (the exectime of T1) closer than the deadline for the whole task. In addition, in executing T2, we must be aware that on completion, there is still T3 (with 5 TUs of work) left to complete. This brings the deadline of the subtask containing T2 even closer. The deadline of this subtask is then 40 TUs. To summarise, in calculating the timing properties of the subtasks we must take into account the work that has completed before the subtask is triggered and the work that is to be done on completion of the subtask.

When considering the task as a set of critical regions, the deadlines and execution times of the individual critical regions should be calculated when assessing the schedulability of the tasks.

The Tests

This section describes the tests that should be carried out on the task set. The tests should be carried out in the order suggested. If any test fails then there is no point in going onto the following tests since they must also fail. There are three sets of tests that should be carried out with the EDF scheduling policy. The first tests check to see that there is enough processing power available to execute the tasks regardless of any need to execute the tasks as indivisible units (as is required for the critical regions). The second and third tests build on the first and check that the critical regions can indeed be treated as indivisible entities such that the deadlines of the tasks are met. In all these tests, the ammended deadlines, as described in the previous section, should be used and not the overall real-time task deadlines.

Test 1 : Raw Processing Power
In the first set of tests we check that there is enough 'raw' processing power to execute all the tasks on a processor before their deadlines. This test ignores the fact that the work of the tasks on a processor may have to be executed as an indivisible unit. In this test the work of a task on a processor can be interrupted and restarted without having to worry about the consistency of shared data on restarting.

The test proceeds by first determining the worst case of task triggerings that can occur on a processor. The worst case is considered to be when all tasks trigger at the same time T_0, and then each task re-triggers at its minimum retrigger time (MRT). As an example consider three tasks with the following characteristics. This is Simple Task Set 1.

1. Execution time of three time units (TUs); deadline of ten TUs from the trigger time; minimum retrigger time of ten TUs from the previous trigger.

2. Execution time of one TU; deadline of four TUs from the trigger time; minimum retrigger time of four TUs from the previous trigger.

3. Execution time of one TU; deadline of two TUs from the trigger; minimum retrigger time of two TUs from the previous trigger.

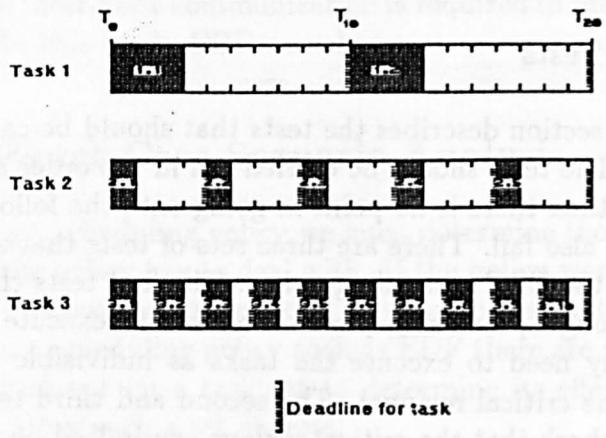

Figure 5.2: Worst Case Triggerings for Simple Task Set 1

We can represent the worst case triggerings of these tasks on a diagram for each task. This is shown in figure 5.2. The shaded part of the diagram represents the execution of the task.

We must now determine if the EDF scheduling policy is appropriate over some period of time. Over what period of time should we consider the task triggerings such that we can be confident that the scheduling policy works for an infinite length of time? It has been suggested we should consider the time period up to the longest deadline of any task. There is no shared data used and hence each task is preemptable. Consequently if there is enough processing power up to the longest deadline to satisfy all those tasks that have deadlines up to that time, then the EDF policy is sound. Using the above example, this means that we should consider all those tasks up to the longest deadline i.e. up to time T_{10}. Upto this time we have one triggering of Task 1, using three Processing Units (PUs), two triggerings of Task 2, using two PUs and five triggerings of Task 3 using five PUs. This adds up to ten PUs required up to time T_{10} and hence the EDF policy would appear to be sound.

The EDF scheduling policy on the triggered tasks up to this deadline will result in the execution trace shown in the Gantt diagram shown in figure 5.3.

Figure 5.3: Execution trace for Simple Task Set 1

Figure 5.3 suggests that the EDF policy works on the task set. However, if we consider the worst case scenario further than the longest deadline we can show that some tasks fail to meet their deadlines. Figure 5.4 shows the execution trace beyond this longest deadline of T_{10}. The figure shows that the second triggering of task 1 only received two processor units up to its deadline instead of the required three. This task therefore fails to meet its deadline and the EDF policy is unsound on this task set.

This result is intuitively obvious. Upto time T_{20} we require 21 processor units to execute all tasks such that their deadlines are met. Instead of chosing the longest deadline as the time frame, we should consider a longer time frame. In the above example, T_{20} was the least common multiple of the deadlines of the task set. We propose then that if there is enough processing power up to the earliest common re-trigger time then the EDF scheduling policy will work on a task set with no shared data. The earliest common re-trigger time is the next time that all the task triggerings occur at the same time.

As an example, consider the following Simple Task Set 2 that consists of three tasks.

1. Execution time of two TUs; deadline of ten TUs from the trigger

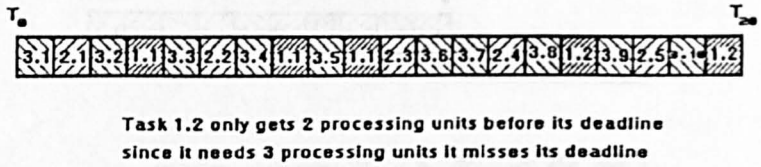

Figure 5.4: Execution trace for Simple Task Set 1 beyond T_{10}

time and minimum retrigger time of ten TUs from the previous trigger.

2. Execution time of one TU; deadline of four TUs from the trigger time; minimum retrigger time of four TUs from the previous trigger.

3. Execution time of one TU; deadline of two TUs from the trigger; minimum retrigger time of two TUs from the previous trigger.

The earliest common re-trigger time of these tasks is at T_{20}. Upto this point we have two triggerings of task 1 requiring four PUs; five triggerings of task 2 requirings five PUs and ten triggerings of task 3 requiring ten PUs. This adds up to to a total requirement of 19 PUs which will fit into the 20 that we have up to the common deadline. We conclude that the EDF scheduling policy will work on the task set where there is no shared data and preemption is allowed.

The above analysis assumes that the minimum retrigger time for a task is greater than or equal to the deadline for the previous task. The minimum re-trigger time for a task may however occur before the deadline of the previous invocation of the same task. If this is the case, then analysis period of the earliest common re-trigger time must be

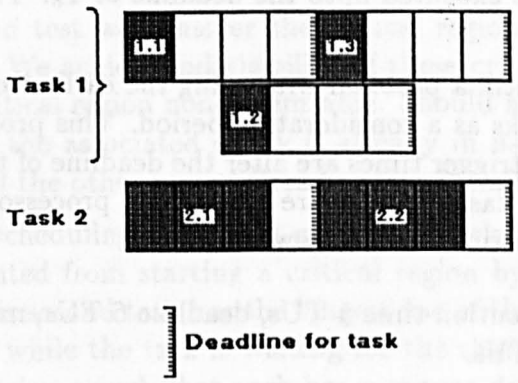

Figure 5.5: Triggerings for Simple Task Set 3

extended slightly. To illustrate this consider the two tasks in Simple Task set 3 that consists of two tasks:

1. Execution time one TU; deadline four TUs from trigger; minimum retrigger time two TUs from previous trigger.

2. Execution time three TUs; deadline four TUs from trigger; minimum retrigger time four TUs from previous trigger.

Assuming that both task 1 and 2 trigger at T_0 and both retrigger at their deadline, we would get the triggerings shown in 5.5.

The first task has a minimum retrigger time less than the deadline of the previous trigger. If we consider the time period up to the earliest common re-trigger time of these tasks i.e. T_4, it suggests that the task set is schedulable with the EDF policy. However, if we consider further than the least common deadline, we see that some tasks can miss their deadlines with the EDF policy. We need a new definition of the consideration period such that it deals with task triggerings that overlap. A suitable consideration period is the second earliest common re-trigger time. With the above example, the earliest common re-trigger time considered one triggering of task one and two, the fact that task one invocations overlap was ignored. In checking upto the second earliest re-trigger time, we take into account any overlapped triggerings of the

tasks. In the example of figure 5.5, the second ealiest re-trigger time after T_0 is at T_8. Upto this time, we have three triggerings of task one and two of task two. This requires 9 units of processing time and hence cannot be executed upto the deadline of T_8. This task set is therefore unsound.

There is still a problem with using the earliest common re-trigger time of the tasks as a consideration period. This problem occurs with tasks whose re-trigger times are after the deadline of the previous task. Consider two tasks that share a common processor and whose temporal characteristics are as follows:

1. Execution time 5 TUs, deadline 6 TUs, minimum re-trigger time 30 TUs.

2. Execution time 5 TUs, deadline 6 TUs, minimum re-trigger time 30 TUs.

Using the earliest common re-trigger time of 30 TUs as the consideration period, shows that the task set is sound. Clearly though, if both tasks trigger at the same time, one will miss its deadline. We need to consider this type of problem in the test for serialisability through a shared processor. With each task whose retrigger time is greater than the previous deadline, for the sake of the static analysis, we assume that the retrigger time is at the deadline of the previous invocation of the task. In the above example, we change the re-trigger times for both tasks to be at 6 TUs. The analysis then shows that there is not enough resource time to meet both the deadlines.

This constraint imposes a worst case on these tasks that is stricter than the actual worst case. It is possible that a task set appears unsound with the constraint applied even though a valid schedule is possible. Applying the constraint will however, always identify those task sets that are really unsound.

This sort of test can also be carried out on any other shared resource in the system. We can carry out the test on all those tasks that use shared data entities or shared peripherals, as well as the physical processor. If the task set fails at this stage then some action must be taken to redesign the task set or allocation scheme. If the task sets pass the tests then the next set of tests can be carried out.

Test 2 : Non-preemptive Scheduling

If the tasks have passed the first test then we known that there is enough time to execute all tasks using their shared resources. The test assumes that the tasks use of the resources may be interrupted by other tasks

with sooner deadlines and the interrupted task restarts from where it left off. In reality this is simplistic. A task's use of a shared resource, whether it is a shared database entity or a physical device, often has to be treated as an indivisible unit of processing.

In the second test we consider the critical regions that use a shared data entity. We achieve indivisibility of these critical regions through making a critical region non-preemtable. Should a critical region need to start but the associated entity is already in use then the region is blocked until the other region already using it finishes.

With EDF scheduling, the worst case for a task T_i is when it is first of all prevented from starting a critical region by another task T_{long} such that this second task has the longest use of the shared data entity. In addition, while the task is waiting for the shared resource, a set of other tasks trigger such that each has a sooner deadline than the first and each needs access to the same shared resource. These later tasks are given priority over T_i by the scheduling policy.

As an example consider the following task set that consists of three tasks each having a single critical region on a shared data entity.

1. Execution time of 4 TUs, deadline of 15 TUs, MRT of 15 TUs.
2. Execution time of 6 TUs, deadline of 15 TUs, MRT of 20 TUs.
3. Execution time of 1 TU, deadline of 5 TUs, MRT of 5 TUs.

The worst case scenario that task 1 will face is if it is triggered just after task 2 is started. This implies a delay in execution of 6 TUs. In addition, the worst case also consists of a triggering of task 3 at the same time as task 1. When task 2 finishes, task 3 will process and then task 1 will get access to the shared data entity. The worst case delay for task one is therefore 7 TUs.

The worst case for task 2 is when it is triggered just as task 1 starts executing and at the same time as task 3 triggers. In the worst case task 2 suffers a delay of 5 TUs.

The worst case for task 3 is when it is triggered just after task 2 is started. This worst case represents a delay of 6 TUs. This worst case is unacceptable since the deadline for task 3 is at 5 TUs; in the worst case, task 3 misses its deadline. We can then conclude that the EDF policy does not guarantee the deadlines for these tasks if preemption is not permitted.

In general for a task triggering T_i there is another task triggering T_{long} that makes the longest use of the shared resource. In addition, there is

a set of other task triggerings, TS, such that each has a sooner deadline than T_i and each is triggered before T_i can start to execute. The worst case delay a task T_i can suffer is

$$\text{Worst Non-preemptive Delay}_i = T_{long}.E + TS.E$$

where $T_{long}.E$ and TS.E are the execution times of T_{long} and each member of TS respectively.

For each use that a task makes of a shared resource, the worst case delay that the critical region on the shared resource makes should be less than the slack of the task (i.e. the spare time between the end of execution and the deadline). If the worst case delay cannot fit into the slack then the task cannot be scheduled using non-preemptive EDF scheduling.

Test 3 : Preemptive Scheduling

In the third test we again consider the critical regions that use a shared data entity. We achieve indivisibility of these critical regions through making a critical region preemtable and providing a back-off and restart scheme for a preempted task. Should a task require a critical region to start, and that task has a sooner deadline that the task currently executing, then the current task is backed off and the pre-empting task is allowed to execute.

The worst case scenario for a task occurs when a task is preempted and backed off just the 'moment' before it has finished with the shared data entity. In the worst case, we have to contend with repeating the complete execution of the task. A further point arises. How many times do we allow a task to be preempted ? If there were not a limit on the number of times a task could be preempted, there is a real danger that the task can be continually interrupted and never complete its work. We can work out the worst case of task triggers to determine the maximum number of times a task will be preempted in the worst case.

Consider the task set described above in test 2. We make the assumption that a task may be preempted and restarted exactly once only. A restarted task becomes non-preemptable and runs through to completion. In this case, the analysis must include the analysis for test 2 i.e. when a preemptable task becomes non-preemptable. The worst case for task 1 is when it is interrupted just as it is about to complete by task 2. Task 1 therefore suffers a further delay of 6 TUs before it can be restarted. The slack for a task must be enough to completely re-execute the task in addition to executing the longest other task.

In general, assuming a task triggering T_i can only be preempted once the worst case delay it will suffer is

$$T_{long}.E + T_i.E + TS.E$$

If this delay is greater than the slack of the task then the task cannot be scheduled using pre-emptive EDF scheduling.

Use of Stale Data and Read Only Access

The analysis described in the three tests does not take into account the fact that a task may be allowed to use stale data. This complicates the analysis slightly since a task using stale data is not affected by another updating the most up to date copy of the same data. The analysis must take into account that these tasks need not be serialised.

For some tasks, there may not be a write as part of the critical region on a data entity. The static analysis can be extended to allow many such tasks to execute concurrently. An example of a complete static analysis of a simple task set is presented in Appendix C.

Summary of the Tests

To test the schedulability of a set of tasks, test 1 above should be carried out using the tasks on each processor. The test should then be carried out considering the other shared resources i.e. shared data entities and other hardware devices. Should the first set of tests suceed then tests 2 and 3 can be carried out for each critical region in each task.

5.3.4 Unsound Task Sets

In some systems an analysis of the task set shows that it is impossible for the scheduling policy to ensure that each task always meets its deadlines. A task set that cannot be scheduled with the chosen scheduling policy is called an unsound task set. The simple answer to dealing with an unsound task set is to provide more processing power. However, there are are several other courses of action that can be taken to finish with a system that meets all its deadlines. These actions are as follows:

1. Identify those tasks that fail to meet their deadlines. Redesign individual transactions to reduce the overall amount of processing required on the task.

2. Identify those tasks that fail to meet their deadlines. Redesign the transaction orderings to find a task precedence graph that is closer to the optimum.

3. Try a different allocation of tasks and transactions to processors.

4. Try the analysis on a different scheduling heuristic.

5. Change the requirements specification and go through the stages of the design methodology in the hope that we can produce tasks with shorter execution times.

6. Use a higher specification processor to reduce the execution times.

The first action implies a systematic 'tweaking' of the individual transactions in the task. The execution time for the task at fault should be reduced and reanalysis of the task set then hopefully shows that the task is now processed before all its deadlines. This action is the least drastic of the options; where possible it should be used in preference to the others.

The second action involves adjusting the task at fault, again in an attempt to reduce the execution time of the task. It may be that in constructing the transaction precedence graph, we have already constructed the optimum graph for the task. If this is the case then the situation can be made worse by changing the orderings of transactions. If however, the execution time of the task can be reduced by reordering some transactions to improve the concurrency with the task, then the scheduling algorithm may work. After generating a new TPG, the analysis should be repeated.

The third action is to try a different allocation of tasks and transactions to processors. The allocation generated by 4.21 could result in a processor that cannot physically process the required transactions before the deadline. If this is the case, then some other allocation is required.

The fourth action is to try a different scheduling algorithm. It may be that where the earliest deadline first policy fails, some other policy may succeed. The whole task system should be tested with the new scheduling policy. The analysis can still use the definition for worst cases described previously in this section although determining whether the worst case is schedulable will be different for non EDF policies.

The fifth action is to change, or relax, the requirements specification. It may be that the definition of those tasks that fail to meet their deadlines can be altered to bring the task to within more suitable execution times. This action can only be carried out with much discussion between the system designers and the party wanting the system. It may be that the requirements can be relaxed without changing the functional characteristics of the task.

The sixth option open to the design should only be used when the other options have been tried. This final course of action is to use higher specification hardware in the design. Given a 'flexible' budget faster processors could be used to reduce the execution times of all the tasks. It may be that the allocation scheme results in particular processors that are heavily loaded. Faster processors could be used in these cases.

5.4 A Run Time Environment

The previous chapters have described how, given a task set we can serialise the critical regions within the tasks to execute correctly the tasks. Each task is presented with a consistent database state regardless of whether it is preempted by some other task. The first two sections of this chapter considered the analysis of a task set to check for its schedulability using a fixed EDF scheduling policy. This section describes a three layered, distributed scheduling mechanism that embodies the need for serialisability of critical sections and that uses the EDF scheduling policy on a sound task set to meet the deadlines of the triggered tasks.

The scheduling mechanism is organised as a hierarchy to simplify its construction. At the lowest level of this scheduling hierarchy is the mechanism used to correctly execute the transactions within a task. This is the transaction scheduler. The middle level of the hierarchy correctly sequentialises the critical regions within a task. This is the critical region scheduler. The highest level is responsible for recognising the task triggerings; this is the mechanism to start a task executing once it has triggered and is known as the task scheduler.

5.4.1 The Schedulers

The Task Scheduler

The task scheduler is at the highest level in the scheduling hierarchy. It is distributed across the processors in a distributed implementation of the system. Each task scheduler is responsible for controlling those tasks that are triggered at that processor. The task scheduler is invoked by one of two actions. The first is receiving an event from the controlled environment. The task scheduler should then start the task by sending a control token to the transaction scheduler associated with the task.

The task scheduler is also invoked when the task has completed.

The task scheduler maintains a table of the tasks that are currently active in the system. It is its responsibility to ensure that if a task is invoked before the previous invocation has been executed then the second invocation does not 'overtake' the first. Each separate invocation of a task is allocated a coloured token; scheduling mechanisms lower down in the scheduling hierarchy will then use these tokens. The task scheduler can also be responsible for gathering statistical information about the relative frequencies of triggerings of the tasks. This information is, together with the next colour token for a task, stored in the task schedulers table.

The Critical Region Scheduler

Each data entity in the distributed implementation has a critical region scheduler. This scheduler is invoked in one of two ways. The critical region scheduler is invoked when the task is about to start a critical region on a data entity and when the task has finished with a critical region.

The critical region scheduler makes the run-time scheduling decisions. To prevent conflicting critical regions from executing concurrently, the critical region scheduler maintains a data entity lock table. When a task needs to enter a critical region, the critical region scheduler checks the lock table. If the lock table shows that the entity is free then the lock for the entity is set as in conventional locking [PBG87], [BG81], [Men79], [Wol87], and the transaction scheduler for that task is invoked. If the lock is set showing that the entity is currently in use, by some conflicting task, then the critical region scheduler decides what course of action is to be taken based on the scheduling policy chosen and the relative deadlines of the respective tasks. It may be that the currently executing task that has control of the data entity needs to be backed off. The mechanism for doing this is described in a later section. If the decision of the scheduler is that the task currently holding the entity should retain it, then the new task is suspended until the entity is again free. If the data entity is currently free then the critical region scheduler passes a token to the transaction scheduler to show that it may begin its use of the data entity. Should a more 'urgent' task need the entity, then the critical region scheduler sends a stop message to the appropriate transaction scheduler to suspend the current task. The critical region scheduler maintains a list of those tasks that are waiting for the associated data entities. When the entity is freed this list is

consulted and a control message sent to one of the member tasks to start it executing.

When a task has finished a critical region, the critical region scheduler receives a control message from the associated transaction scheduler. The lock on the data entity is then removed. At this point, the critical region scheduler handles the propagation of any updates made to the entity, to the other copies of the entity used in the same task.

The Transaction Scheduler

The transaction scheduler is invoked when it receives a control token from the task scheduler. The control token tells the transaction scheduler that it must start executing the given task. (So that the transaction scheduler knows the order of execution of the transactions, it has some representation of the transaction precedence graph for the tasks. Also the transaction scheduler maintains a list of all the transactions in the task.) When it receives a control token from the critical region scheduler, the transaction scheduler starts the first (root) transaction in the task.

Each transaction has an identified set of successor transactions and an identified set of predecessor transactions. When a transaction finishes, the predecessor entry in each of its successor transactions is changed to show that the predecessor has finished. When the list entry for a transaction shows that all its predecessors have completed, the transaction scheduler will execute the transaction. This transacti on execut io

n mechanism is effectively an implementation of an executing petri-net.

The transaction scheduler may at some time receive a stop message from the critical region scheduler. When it does, the transaction scheduler must restore its transaction list entries for the current task to a suitable point and then wait to be re-started by the critical region scheduler. When the transaction scheduler finishes a critical region, it sends a control token back to the critical region scheduler. When the transaction scheduler finishes executing the task, it sends a control message to the critical region schedulers. This message consists of the updates to data entities and also to the task scheduler an aknowledgement that the task has finished.

5.4.2 Managing distributed/replicated data

We have already established that each task needs to be presented with a consistent database state and that on completion, each task must also leave the database in a consistent state. This requirement was refined by the introduction of a critical region. A task consists of interlocked critical regions each of which is presented with a consistent state of the associated data entity and each of which leaves this entity in a consistent state. The execution environment for the real-time tasks needs a mechanism for ensuring that critical regions are treated as atomic units of processing as far as a task is concerned. This requirement is met by the task, critical region and transaction schedulers.

Each task has its own local copies of the data entities that it requires. If a task has multiple concurrent reads of an entity, then in an ideal implementation of the system, there is one copy for each read. Any updates that a task makes to an entity are only applied to the task's local copies of the entity. The critical region and transaction schedulers ensure that critical regions are executed as atomic execution units and that updates are correctly propagated to the local copies of data entities used in other tasks.

When a task wishes to 'enter' a critical region on a new data entity, the transaction scheduler sends a request to the controlling critical region scheduler. If the request fails, i.e. the data entity associated with the critical region is in use by some other, more urgent, task then the critical region scheduler replies with a wait message. The transaction scheduler then suspends this task until the entity becomes free. If the request is successful, i.e. the data entity is not in conflicting use, the critical region scheduler responds with a confirmation. The transaction scheduler then starts executing the new critical region.

On successfully starting a critical region, its transactions are executed. If a transaction updates the associated data entity, the changes are immediately applied to the local copy of the entity 'owned' by the task and used by that transaction. On completion of the transaction, the transaction scheduler propagates the updates to the other copies of the entity also owned by the task. On completion of a critical region, the transaction scheduler sends any updates made to the associated data entity to the critical region scheduler. The critical region scheduler propagates these updates to all copies of the data entity that are owned by different tasks. The critical region scheduler then makes the data entity available to other tasks.

A critical region of a task, A, may be suspended if a more 'urgent' task

,B, requires the use of the data entity. Two cases must be considered; these are firstly when the interrupting task, B, simply reads the data entity and secondly, when task B updates the data entity. Let us assume that task B does not update the data entity. Since any updates from task A will not have reached the copies of the entity used by task B, task B has a consistent copy of the data entity and may go about its work. On completion, task A can be restarted; since the entity has not been updated, task A can restart from exactly the place it left off. The overall effect of this is that task B uses slightly out of date data.

Let us now consider the case when task B, the interrupting task, updates the shared data entity. On starting the critical region in B, the task has a consistent copy of the data entity since the partial updates from A do not reach it until the critical region on A has finished. Task B can go about and update the data entity. On completion of the critical region, task B sends the updates to the data entity to the critical region scheduler. This then sends these updates to the copies of the data entity used by other tasks. As a result, the partial updates to the shared data entity that task A had applied are now overwritten by those of task B. If task A was now restarted from exactly the place it left off, the work it carried out after the interrupt would be inconsistent with that before. Consequently, if a critical region is interrupted and the associated data entity is updated then on restarting the task, we must ensure it begins its work from the beginning of the region.

Restarting a critical region may involve backing off other critical regions as was shown in Chapter 3. In an implementation, either state saving of dependent data entities or multiple back offs could be employed to ensure completely consistent execution of an interrupted task.

The provision of each task with its own local copy of every entity does imply a large (but determined) overhead in terms of propagation of updates. Each task having its own local copies of the shared data entity does bring other benefits though. One of these is that a task can use stale data. Suppose we have two tasks. The first will update a shared data entity and the second reads the entity. We can allow these two tasks to execute concurrently providing the updates from the first task are not applied to the data entity in the second task until this task completes. The updates that should be applied to a task's copy can be queued until the task has finished using the copy. This technique increases the possible concurrency and allows a reading task to use relatively 'stale' data.

In propagating the updates of a critical region, the critical region scheduler may have to employ some recoverable commit protocol to ensure

that all remote sites receive the updates correctly. In a real-time syste, we may however be able to relax this requirement somewhat. For data that has low integrity we may be able to send updates without some commit protocol. Example low integrity data sources are those that change very quickly. If a ermote site does not receive the correct update, then it is not long before it receives another image of the data. The chances of this being incorrect also are small. For high integrity data the critical region scheduler must employ some commit protocol. to keep overheads to a minimum high integrity data should have low volume in the system and low integrity data can have high volume.

5.4.3 Recovery and Failure

The transaction model guarantees that a transaction either succeeds and its results are made permanent on the database, or that a transaction fails and it has no effect on the database. In a fault tolerent environment, the loss of transactions through processor failure can be prevented by placing redundant copies of transactions on separate processors. Should the transaction scheduler detect that a processor is failing (or has failed) in some way then it ignores the primary copy of the transaction and switches to use the redundany, back-up, copy. The transaction lists maintained by the transaction scheduler for a task indicate the location of the copies of each transaction. The transaction scheduler can then use those copies on processors it knows are functioning correctly.[1] Further problems can result dependending on when the particular processor that executes the primary copy of a transaction fails. If a processor fails after the transaction scheduler has sent a control message to start a transaction on it, the transaction may never complete. The transaction scheduler must have some way of knowing that the processor has failed. Of course, processor failure affects the determinism of the real-time system; the probability of failure should be incorporated into the worst case analysis.

5.4.4 Replication of the Scheduling Components

The three scheduling components may be replicated and distributed to remove bottlenecks and potentially improve resilience. We now describe three schemes with different configurations of task, critical region and transaction schedulers.

[1] Further issues of fault tolerence are discussed in Chapter 6.

Scheme 1 : Multiple Transaction Schedulers

In the first scheme, each task in the system has a set of processors, or a 'cluster', dedicated to it. Each processor is a 'node'; each node has a set of transactions and copies of the appropriate data entities used by those transactions. The transactions within a task are allocated to the processors used by that task, to maximise concurrency. Associated with each cluster is a transaction scheduler. This is responsible for scheduling the transactions for that task only. When a transaction updates a shared, replicated data entity, the transaction scheduler propagates the updates to the other copies used by that task (i.e. within that cluster).

A single critical region scheduler is used. When a transaction scheduler recognises that the task is to enter a new critical region, it sends a request to the central critical region scheduler. This coordinates the actions of the distributed transaction schedulers. When the end of the task is reached, the task scheduler propagates any updates to the copies of shared data entities used by other tasks (i.e. in other clusters).

The block diagram of figure 5.6 shows this configuration. In this example, the system consists of three tasks. The first task consists of three transactions and two copies of the shared data entity A. The second task consists of two transactions and two copies of data entity A. The third task consists of one transaction and one copy of the data entity A. There is one critical region scheduler resposible for controlling access to data entity A.

Multiple Transaction and Critical Region Schedulers

The single critical region scheduler in the first scheme may become a system bottleneck. In addition to replicating the transaction schedulers, we can also provide more than one critical region scheduler in the distributed system. Each critical region scheduler is responsible for coordinating the access to a static set of database entities. The transaction scheduler will send the requests to enter and leave a critical region to the appropriate critical region scheduler. The block diagram of figure 5.7 shows this arrangement.

The main concern with this distribution scheme is how to partition the database into a set of entities, each set controlled by one critical region scheduler. It is a sensible requirement that these partitions result in an even loading of the individual critical region schedulers. Analysis of the database requirements helps in deciding what the database responsibilities of each critical region scheduler are.

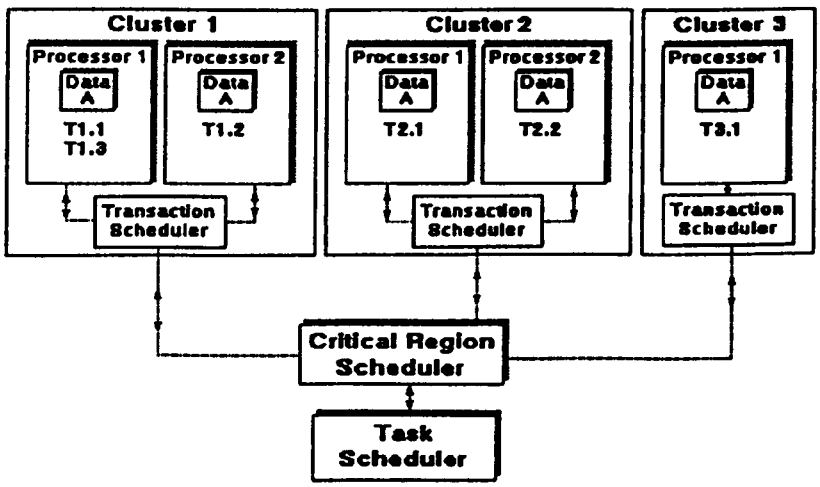

Figure 5.6: Distributing The Transaction Scheduler

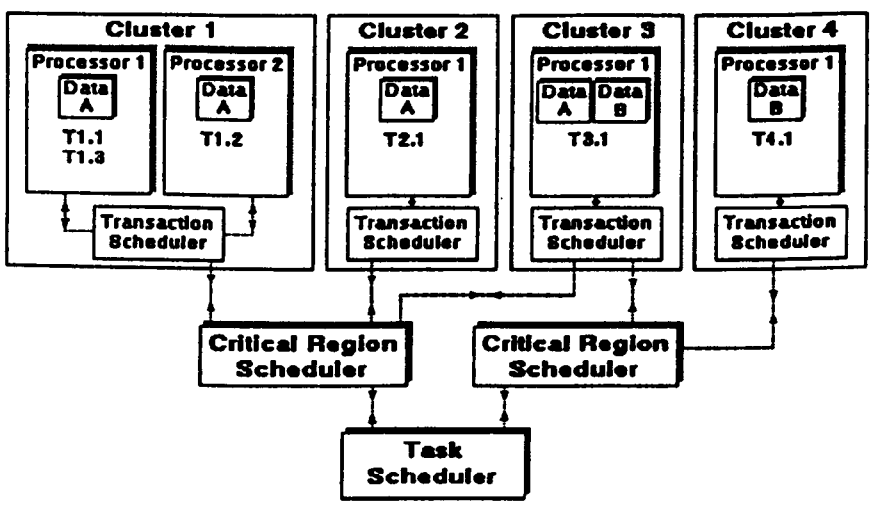

Figure 5.7: Distributing the Transaction and Critical Region Schedulers

Multiple critical region schedulers need to communication with each other. Suppose a task is currently executing a critical region on shared data entity A, that is completely contained within some other critical region on data entity B. Suppose the access to the entities is controlled by two separate schedulers and that the critical regions within the task are 'd epen dent' on each other. If the task is preempted and must back off its critical region A, then we must also back off critical region B. This implies that the critical region scheduler for A must inform the critical region scheduler for entity B that the task has backed off its critical region on B.

The placement of the transaction and critical region schedulers is an important decision. For each task, there is one transaction scheduler responsible for executing the transactions of that task in the right order and for each entity there is one critical region scheduler responsible for serialising the concurrent, conflicting accesses to the entity. In order to reduce the amount on inter processor communication in an implementation, the transaction scheduler for a task should be placed on that processor which is resposible for executing the most transactions of the task. Similarly, the critical region scheduler for an entity should be placed on that processor which has largest proportion of transactions wishing to access the entity. These placement rules can be seen in the Ship Control System of Appendix B.

To demonstrate how the hierarchy of schedulers works in an implementation, the reader is referred to the example execution traces in Appendix D.

5.4.5 Scheduler Overhead

For any dynamic scheduling decisions in a real-time system to be effective, the scheduling mechanism must take into account its own activity. In some dynamic real-time scheduling such as in the Spring Kernel [SR89], the scheduler is considered as a periodic task and is always invoked even if it has no actions to perform. This is fine but it does introduce an unnecessary overhead when no tasks have triggered.

In our real-time scheduling mechanism, the scheduler is very simple. The critical region scheduling mechanism makes its decisions based on the deadlines of two tasks in conflict. The overhead imposed by the scheduler is minimal. We must, however, determine the nature of this overhead.

In some dynamic real-time scheduling mechanisms, the scheduler is

distributed, and in order to make its decisions, it must communicate state information with other, remote, parts of the scheduling mechanism. This communication imposes a further overhead in addition to the actions of the local scheduler.

In our scheduling mechanism, the critical region schedulers are responsible for controlling access to a subset of the data entities. A critical region scheduler is invoked each time a task wishes to start access to a new critical region and each time the task leaves the critical region. On entry to the critical region, the critical region scheduler may have to make a scheduling decision if some other task already has access to the required data entity. When a task leaves a critical region, the critical region scheduler must propagate any updates to copies of the data entity used within the task.

Each task has an overhead for scheduling decisions and propagation of updates. This overhead must be added to the worst case execution time for the task and, as such, must be considered in the static scheduling analysis. The overhead is described by:

$$\sum_{\text{No critical regions in task}} ((CR_o + 2CM) + (CM + \sum_{\text{No local copies}} PO_l) + \sum_{\text{No remote copies}} PO_r)$$

CR_o = maximum overhead in critical region scheduler
CM = overhead in sending a communication message between scheduling components
PO_l = time to propagate an update to a local (in same task) copy of a data entity
PO_r = time to propagate an update to a remote (in other task) copy of a data entity

The first term represents the entry of the task into a new critical region. There are two control messages for each critical region entry. The first is from the transaction scheduler to the critical region scheduler and represents the request to enter a critical region. The second is the reply from the critical region scheduler. The rest of the first term represents the maximum overhead (CR_o) of the critical region scheduler executing the EDF algorithm to determine what to execute next.

The second term represents the finish of a critical region. The CM represents the control message from the transaction scheduler to the critical region scheduler to indicate that the task has finished with a critical region. The second part of the term represents the overhead in

propagating any updates to each of the local copies of the data entity associated with the critical region.

The final term represents the overhead in propagating any updates to each copy of the changed data entity used outside of the task. This is the overhead incurred when a task completes.

5.5 Other Overheads

In addition to the overheads described in the previous section, there are several other overheads that must be considered in calculating the execution time of the transaction. In a disk based system, there is the latency involved in accessing the database entities from the disk. This latency can be reduced and made more deterministic by using main memory database technologies [Eic89]. The latency of propagating update messages to remote copies of replicated data entities must also be considered. Estimating the latency of a network is a harder problem than for the disk. This latency can be reduced by using fewer copies of data entities.

5.6 Conclusions

In this chapter we discussed the problems of scheduling real-time tasks. Both static and dynamic approaches were considered. It was stated that 100% confidence cannot be placed in a dynamic scheduling mechanism unless some static analysis of the task set is carried out prior to execution of the system. This chapter presented a means to consider statically a task set without known task trigger times. The aperiodic tasks in the set are converted into period tasks by considering a worst case scenario when every task continually re-triggers at the earliest possible times. This worst case scenario may then be considered statically. The effectiveness of dynamic scheduling policies such as EDF can be determined.

In addition, this chapter described a replicated, hierarchical run-time environment that ensures each transaction within the real-time tasks is presented, and leaves, consistent database states. Examples of the operation of this run-time environment are described in Appendix D.

Chapter 6

Evaluation of the Work

Uranus, the Magician

6.1 Introduction

In this chapter, we present an evaluation of the method described in the previous chapters and compare aspects of it with its equivalent in other, established, real-time design methodologies. This chapter highlights some of the advantages and disadvantages of the new method compared with others. The chapter begins with an overview of the new method. Each step of the method is considered in turn. Following this, the real-time execution platform described in Chapter 5 is discussed. We then consider how the method and execution platform treat the problems introduced by the need for reliability against failure.

6.2 Evaluation of the Methodology

To judge the success of a design methodology, it is important to consider all aspects of the methodology and compare these aspects with the equivalent in existing, proven, methodologies. Appendix B shows that the transaction based design method described in previous chapters can be used to guide the design of a real-time system. It is important to understand how the method can be integrated with existing methodologies so that they may complement each other. In this section, we present an overview of the design methodology and discuss the relative merits of each stage.

6.2.1 Overview of the Method

The real-time design method has the following stages:

1. Identification of real-time triggers.
2. Decomposition of systems into subsystems.
3. Decomposition of subsystems into tasks.
4. Initial real-time database design.
5. Decomposition of tasks into transactions.
6. Representation of database requirements using DDR (data dependency ring) notation.
7. Representation of tasks using TPG (transaction precedence graph) notation.
8. Allocation of transactions and data entities to processors.
9. Conversion of aperiodic tasks to periodic and static analysis of schedulability.

According to [Ben88], and [Gom86] there are two distinct phases to the design of real-time systems. The first section is the planning or requirements analysis and specification phase. The second is the design or development phase.[1] The real-time design methodology described in this work assumes that the first phase of the design is complete. The method does not address the problem of generating unambiguous and correct system specifications.

6.2.2 Real-Time triggers

The transaction based design methodology relies on the identification of the external events in the controlled environment that require some response from the real-time computer system. This is the case for all real-time design methodologies. Methods such as in [YC78], [MP84] and MASCOT [Bat87], [Jac84] identify those events in the real-time world that require some action from the computer control system. In our method, these events, or real-time triggers, are represented on a

[1] Indeed, these phases should be present in the construction of any computing system

modified context diagram. This diagram shows the boundary between the controlled environment and the controlling computer system. The modified context diagram shows the external events more clearly than, for example, the preliminary design diagrams of MASCOT; the diagram shows only the triggers and not the particular parts of the control system that respond to them. The addition of the periodic trigger symbol allows the designer to distinguish between different types of triggering event. The context diagram also defines the output control actuators through which the computer system controls the environment.

6.2.3 Subsystem/Task Decomposition

It is well recognised that decomposition pervades the entire engineering process and has great influence on the design of real-time systems [BW89]. There are many recognised methods of decomposing, or recognising related activities of, an application. Among these methods are the functional decomposition enforced by modular programming constructs; information hiding [Par72]; maximising cohesion and minimizing coupling [Som89] and partitioning to minimize interfaces between modules.

In our method, an initial functional division of the application into sub-systems is carried out. Like the module subdivision of MASCOT, this is largely dependent on the experience and skill of the designer [Ben88]. The subsystems of our method serve only to decompose the application into more manageable units. After further breaking them into real-time tasks, the sub-systems are not considered further.

After decomposing the application into subsystems, our method further decomposes each subsystem into a number of independent, real-time tasks. The definition of a task is that it is that processing required in response to a independent trigger from the outside world. Since each trigger is separate and independent, each corresponding task is asynchronous with respect to all other tasks. This method of further decomposing a real-time system has been used sucessfully in the DARTs design approach [Gom84]. Having a separate task defined as all those activities to be executed in response to an independent trigger results in a high degree of functional cohesion within the task. The communication between tasks should then be kept to a minimum. Should a task need to communicate with another then this implies that the two tasks are functionally related. This in turn suggests that the two should be combined into one task.

A consequence of this very high degree of functional cohesion within a

task and very low coupling between tasks, is that there is little need for tasks to explicity communicate with each other or synchronise their actions. As a result, our design methodology has no explicit inter-task communications primitives. Implicit inter-task communication is carried out, however, through the use of shared data entities. This is similar to the State Vector Inspection (SVI) of JSD [Jac83], [Sut88] and [Cam86]. In SVI any process may read a shared data item but only one, the owner, may update the entity. In our method any process may update the shared data; there is no concept of an 'owner' of the data. There is no equivalent of the explicit datastream between two tasks in our method.

In many concurrent systems, there is a need to synchronise actions. Our model of the real-time task assumes that there is always a particular invocation of a task that is 'enabled' (i.e. ready to trigger) and waiting for the event to occur in the environment. This is regardless of whether or not the task is already executing for a previous triggering of the event. This implies that there could be multiple invocations of a task active at any one time. It may however, be desirable for there to only be one invocation of the task active at any one time and for multiple, successive, triggers of the same event to be queued up and dealt with when the current invocation of the task is complete. This corresponds to MASCOT channels between the environment and the task. Alternatively, it may be required that if a event triggers during the execution of a previous invocation of the task, then the successive triggerings are ignored until the current invocation has completed. As it stands, our method does not include queues of triggers or throwing away of triggers if a task is not immediately ready to process them. However, extending the method to deal with the cases just described is not difficult and could be considered an implementation issue.

The task decomposition criteria proposed by our design methodology uses a functional decomposition similar to that of DARTS [Gom84], Higher Order Software [HZ] and Structured Analysis/Design [YC78] [DeM78] [MP84]. This functional decomposition addresses both the problems of splitting the application into modules and determining the place of concurrency in the design. The decomposition yields modules with a high degree of temporal cohesion; that is the task contains all those activities that are executed at approximately the same time based on an event taking place in the environment. Temporal cohesion is not considered a good decomposition criterion by methods such as in Structured Design [Gom84]. However, coupled with the high degree of functional cohesion that results, aiming for a high degree of temporal cohesion results in well defined, self contained tasks. Indeed, temporal

cohesion is used in methods such as DARTs [Gom84].

6.2.4 Database Design

The methodology states that the real-time data entities to be included in the real-time database must be decided. Although glossed over by the method, this stage does have a very important effect on the overall design of the real-time system. The method implies that the designer must have a 'rough idea' of the structure of the database: the designer must name each of the entities that are of concern. The transactions that form the real-time tasks are then defined in terms of actions on these entities. The order of execution of transactions is partly determined by conflicting accesses to the shared data entities. Consequently, if the designer does not specify the real-time data entities to a small enough grain size, then the transactions are unnecessarily serialised. If the grain size is too small, very large degrees of parallelism may result since conflicting transactions become rarer. It may not be practical to provide enough physical processors to implement this parallelism and the overhead in implementing it sequentially on a single processor may be worse than having a larger grain size and sequential transactions to start with. Some skill and experience on the part of the designer is therefore necessary to describe the entities at the appropriate granularity.

The granularity also has an effect on the size of the transaction. If an entity is a large and complex data structure then a transaction that is required to update it might also be large and complex. Alternatively, if the real-time entity represents a single record in a table, then the transaction might be a very small and simple process. The designer is faced with a trade off between the amount of work a transaction performs and the overhead necessary to implement parallel transactions on a sequential machine.

The design methodology does suffer a limitation that has a direct bearing on the real-time database. Since the method relies on a static analysis of the database requirements of the real-time tasks in order to generate control flow and determine the schedulability of a set of tasks, there is no run-time creation of shared data entities. A task may create entities at execution time but these are local to the creating task only; no other task can directly access the new entity. If tasks are able to create new shared entities at run-time then the data dependencies between tasks will change and the determinism of the system be affected.

6.2.5 Transaction Decomposition

The transaction decomposition used within the methodology is straightforward and intuitive. The designer identifies the transformations that are required of each of the database entities. Each transformation then becomes a transaction. This is very much a functional approach to the design of the task. The designer identifies the results that are required and not the order in which the results should be evaluated. Each transaction has a set of input parameters (the data entities that it reads) and generates one output parameter (the data entity that it updates). The transaction is itself very much like a traditional function within a programming language.

The decomposition into transactions should fall neatly into place through the identification of the updates that are required on the data entities. This decomposition does not however consider other issues such as the amount of control that is required between the resulting transactions to ensure the consistency of the database. Although this control is generated automatically in later stages of the methodology, there are good and bad transaction decompositions; the bad decompositions result in excessive serialisation of the transactions. The methodology should pay more attention to the decomposition of the task into transactions such that the serialisation of transactions is kept to a minimum.

6.2.6 DDR Notation

The method includes a diagrammatic notation for expressing the data dependencies between tasks. Since the notation is intended to capture all the information necessary to generate automatically the transaction precedence graphs, the notation also includes constructs to indicate any enforced control flow that the designer requires in addition to that imposed to serialise conflicting transactions. This is probably one of the main disadvantages of the diagrammatic notation. Not only does it express the data dependencies between tasks but it also tries to capture the enforced control within a task. The data dependency rings can consequently become messy and cluttered. Perhaps expressing the enforced control flow should be abstracted away from the DDRs and described by partial precedence diagrams. These can then be used together with the DDRs to complete the picture and generate a full transaction precedence graph.

Depending on the granularity of the real-time data entities, the size of the data dependency rings (where size is expressed as the number of

tasks and transactions on the circumference) can become very large. The ring notation can therefore become cumbersome for complex systems. In its defence, however, where it is possible to decompose a data entity, we can draw a hierarchy of primary, secondary and tertiary data dependency rings. At each level of the hierarchy, the number of transactions using the entity decreases. As an example consider a track table. At the top of the hierarchy, we might have two tasks that use the table. If the table is to be constructed from two subtables, one for radar tracks and another for sonar tracks, we could draw two data dependency rings each containing one task (one using radar data, the other using sonar data). Another way to introduce hierarchy into the diagrammatic notation is to use the hierarchy found in the subsystems and tasks. At a top level data dependency rings can be drawn with the subsystems on the circumference. Further down the hierarchy we might place the individual tasks around the circumference. At the lowest level we can express the individual transactions that use the entity.

The data dependency ring notation does not differentiate between different types of data. Often, in real-time systems, there is both time discrete data (i.e. data that has a constant value for a non-infinitisimal period of time) and time continuous data that is ever changing. In an implementation of a real-time system time continous data is represented by time discrete data entities through the sampling of the data source at regular intervals. 'Logging systems' that track and record the 'history' of some environment are examples of such systems. The diagrammatic notation does not distinguish this property of the data entity. As an extension to the notation, updating a time continous data entity could be represented by double arrowed writes on the DDR for the entity. An example is shown in figure 6.1. The transaction in task number one has a double headed arrow showing that this is a logging task for the data entity. The double headed arrow construct is also found in time continuous data transformations in [WM86].

6.2.7 TPG Notation

The transaction precedence graphs created as part of the design methodology show the complete control flow through a task from the triggering event through to an optional response back to the environment. The control flow is necessary to prevent conflicting transactions from executing concurrently and to ensure that the application requirement constraints are met. The main problem with the notation used to express the transaction precedence graph is that, as described, it is not hierarchical. This comes about from the definition of the transaction

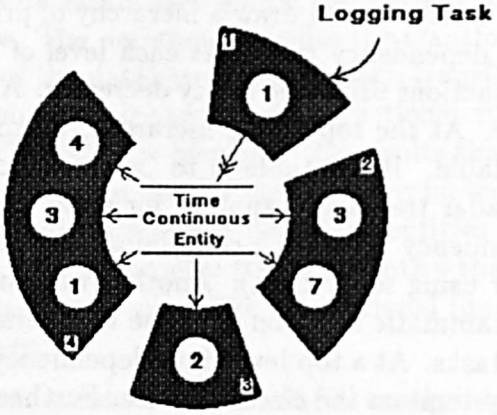

Figure 6.1: Representing Time Continous Data Entities

as an indivisible unit of processing. Consequently, for some real-time tasks that consist of many transactions, the corresponding transaction precedence graph may be very large and cumbersome. We can, however, consider the graph as being composed of collections of subtasks. A graph of these subtasks is less complex than the equivalent graph of transactions and may be a useful tool in considering the control flow through a task in the design stages.

It is desirable to have some criteria for judging the quality of a given transaction precedence graph. In an ideal world, each transaction precedence diagram would have some maximum 'width'. Chapter 4 stated that the closer a graph's width is to this maximum the better. However, as will be explained, finding this width for a given task set is a difficult problem. Another way in which the transaction precedence graphs may be considered is as a collection of D-structures [BJ66], [BS72], [Har80] extended for concurrency [NS90]. In our transaction precedence graph the D-structures are either simple transactions or constructed from other D-structures each of which may be either a sequence of D-structures, a selection D-structure or an interation D-structure. Each D-structure has only one entry and one exit point for control. In ensuring that the transaction precedence graphs satisfy these rules, the transaction precedence diagrams avoid the equivalent

of 'spaghetti-programming'.

The transaction precedence graphs are generated automatically from partial ordering information in the

fo

rm of application requirements constraints and from the data dependencies among the component transactions. As stated in Chapter 4, finding the best graph as well as the maximum width for a task set, is a difficult problem. Several heuristics were described to help generate graphs close to the best case although the solution to the problem is still computationally intensive.

A final problem with the definition of the real-time task is that it is triggered by a single, independent trigger from the environment. This definition excludes those tasks that are triggered by the occurrence of multiple events. This sort of task can always be implemented using the single trigger definition of a task. A special synchronisation task for each of the multiple triggers, updates some shared data entity to show that the associated event has triggered. When one of these synchronisation tasks recognises that all of the required events have triggered by checking the state of the shared data entity, it sends a single trigger to the main task that waited for all the individual triggers. To represent this situation on the transaction precedence graph, we could use multiple triggering boxes as in figure 6.2. The task is executed when each of the three events have triggered.

Even this extension to the notation does not encompass all the possible triggering combinations. For example, what about the task that should be executed if two out of three different events both trigger ? Of course we can implement this with the single trigger synchronisation tasks but the notation doesn't help the representation of the problem. At implementation time we could have three separate tasks for each of the triggers. The task will set a shared flag and then test the flag of the other tasks. If these flags have been set then the main task (to be executed when all events have triggered) is executed. This, however, implies that the main task has to be replicated three times which is not desirable for complex systems. An alterative is for the main task to poll the event flags. This is again undesirable. A suitable solution would be to make the real-time database 'active' rather than passive. Associated with each entity could be a list of tasks to be executed if the entity is updated. As soon as the entity is changed, the tasks in the list are executed. This is a better solution than the tasks polling the entities and has been employed in other real-time databases such as the Diomedes database machine.

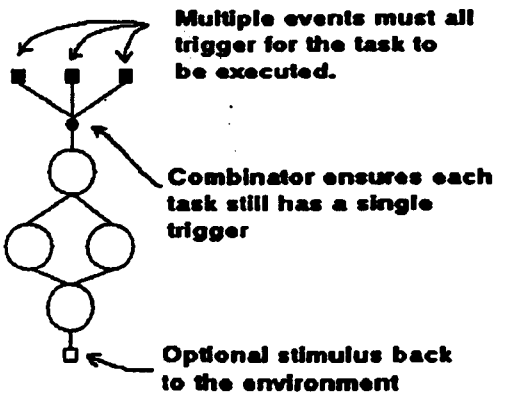

Figure 6.2: A Task with Multiple Triggers

6.2.8 Allocation Schemes

The allocation scheme used by the methodology is simple but reasonably effective. The initial allocation of logical processors to the cluster for each task is determined by the 'width' of the task. Wide tasks exhibit more concurrency and therefore require more processors. After allocating logical processors to the cluster, the methodology describes a way in which to assign the work of the logical processors to the physical processors in a network. A simple way to do this allocation is to give each logical processor in each cluster a number and assign its work and data entities to the physical processor with the same number. If there are more logical processors than physical processors available then we obviously have to 'double-up' on the assignment of logical to physical processors. This scheme results in the work of the first logical processor of each task being assigned to the first physical processor; the work of the second logical processors of each task being assigned to the second physical processor etc. On first consideration, this scheme is reasonable. It is assumed that in general, the tasks do not trigger all at the same time; the physical processor set can then be dedicated to achieving the best use of concurrency for each task as it triggers.

This allocation scheme may, however, result in poor distribution of data. Allocating the work of the system based purely on the concur-

rency that may be achieved within the task can result in a copy of each data entity being on every physical processor. A fully replicated database is inefficient in terms of having to propagate updates to the replicated copies of the entities. A better allocation scheme takes into account the locations of data entities that have already been assigned to a processor. The allocation scheme described within the method and considered in Chapter 4 attempts to allocate the work of logical processors to physical processors that already have the required data entities allocated. A further constraint ensures that the work and entities of a logical processor are placed on a different physical if the first physical processor considered has the correct data entities but which also has some transactions within the same task that are to be executed concurrently. This ensures that concurrency within a task is still recognised but the logical processors are placed on the physical processor with the best allocation of data entities. A result of this allocation scheme is that the load (in terms of numbers of transactions and data entities) that is placed on each physical processor is not even. Some processors will have many, often used, data entities allocated to them with all those transactions that access the entities. Other processors might have little used entities and few transactions. However, all design is about 'trade-offs'.

One of the drawbacks of a static allocation scheme is that no benefits can be gained from spare processing capacity in one processor when another is experiencing a 'transient' overload [BW89]. This problem has been addressed in some systems [SR89] by dynamic placement of sporadic tasks. The disadvantage of these systems is that there is still a possibility of missing deadlines as a processor is located to execute the newly triggered task. A static placement in conjunction with a worst case static analysis of the load on each processor can be used to gaurantee all deadlines.

6.2.9 Static Analysis

The static analysis to test for the schedulability of a given task set relies on the existence of either known start times for the tasks (as in the periodic case) or the minimum interval between sucessive triggerings of the same tasks (for the aperiodic case). This minimum interval was described as the minimum repeat time (MRT) of the task. These periods and MRTs are used to determine a worst case scenario of task triggers. The suitability of a given scheduling policy is determined in this worst case. If the policy works in the worst case, it is guaranteed to work in all other cases that occur at run-time.

One of the major problems with the analysis method proposed is in determining the worst case. In most cases, the MRTs for the aperiodic tasks are evaluated by considering the physical properties of the mechanism that causes the trigger for the task. For example, a radar device has a definite time before the next set of data is available. For some tasks it may not be possible to determine the MRT. If this is the case then some estimate must be made. This estimate can of course have a great bearing on the construction of the worst case of task triggerings and the static analysis of the worst case that follows this. Should the estimated MRTs be too small then the analysis may incorrectly show that the task set fails with a given scheduling policy. Similarly, should the estimated MRTs be too large then the analysis may incorrectly show that the task set works with a given scheduling policy. It is more 'dangerous' to make the second mistake; a system design may result whose static analysis showed its correctness but that does not actually meet the deadlines of all tasks in the absolute worst case.

The static analysis was split into three parts. The second part determined the worst case delays that a task would suffer if tasks with sooner deadlines triggered at the same time and these tasks were run first. The third part determined worst case number of times that a given task could be preempted and thus have to start from the beginning of its work. These last two parts of the static analysis are reasonably straightforward. In the first part of the static analysis, we determine whether there is enough 'raw' processor time to execute a given set of tasks before their deadlines. In addition, we can repeat the test to determine whether there is enough time to execute each task 'through' its data entities. (This test is, however, superseded by testing the 'raw' processor power). A problem exists with this first part of the static analysis. The test relies on being able to find the soonest common retrigger time of all the tasks given that they have all simultaneously triggered at T_0. For a large task set, with large MRTs, this earliest common re-trigger time can be a very large number indeed[2]

A further problem of the static analysis is that it does not take into account the fact that there is a probability of failure of a transaction. The absolute worst case scenario for a transaction includes the delays introduced by preempting transactions but ignores delays introduced

[2]In the naive method to calculate the common re-trigger time used by the CASE tool described in Appendix A, the largest MRT is found; a count is incremented from 0 in steps of this largest MRT. At each step, all the MRTs are tested to see if they divide exactly into the count. If they all divide, then the count is the earliest common re-trigger time. For a large task set, this algorithm can take a very long time.

through the failure, both of the preempting and preempted transaction. Some estimation of the probability of transaction failure should be given and this worked into the worst case execution times for the transaction.

Although the static analysis has problems, it does provide a starting point for the systemmatic schedulability evaluation of the design. The analysis is the part of the methodology that recognises that in hard real-time systems tasks must complete before their deadlines. Other methodologies do not treat the temporal characteristics of the tasks in such depth. Even when the first stage of the analysis is computationally intensive, in determining the worst case of task triggerings, we effectively reduce the tasks to a set of periodic tasks. Other real-time scheduling techniques specifically for periodic tasks may then be applicable, for example [Mar82]. (The complexity of the scheduling analysis proposed by [Mar82] for m processors and n tasks is $O(m^2n^4+n^5)$.)

6.3 Evaluation of the Execution Platform

Chapter 5 described an execution platform for the real-time tasks generated by the real-time design methodology. This consists of a hierarchy of scheduling mechanisms corresponding to the hierarchy of 'execution components' in the real-time system, namely: the task, the critical region and the transaction. The action of each of these scheduling components is well defined; each component can communicate with the schedulers at the next (or previous) level using simple messages.

One great advantage of the scheduling hierarchy is its own modular construction. It is easy to build an execution environment for different distributed systems using different combinations of the task, critical region and transaction schedulers. This should not only benefit the performance of the system, through concurrency within the execution platform, but also resilience to failure can be improved by replicating the scheduling components as necessary. The functions of each scheduler in the system are relatively simple but, when considered together, they provide a powerful tool that ensures that the database consistency is preserved at all times and that real-time deadlines are also met.

Some important aspects of the execution platform have not been considered. Among these is the internal structure and access mechanisms of the real-time database. This is considered as an implementation issue and is not important at the design stage. There are, however, some important constraints placed on how the data may be actually accessed. The latency for access to the local data must be well defined so that

it may be included within the excecution time for the transaction during the static analysis stage of the method. Main memory database architectures [Eic89] can help to ensure time constrained accesses.

The execution platform is very optimistic in that it assumes that no failures occur within the system. Transactions can, and do, fail in a computer system. The execution platform can handle the failure of a transaction by adjusting the transaction schedulers lists of transaction states so that the transaction is restarted. In addition, the database must be restored to the state just before the start of the failed transaction. This can be achieved by copying the partially changed data entities from a consistent copy either within the same cluster or from a cluster belonging to some different task.

6.4 Reliability and the Method

Computer systems are increasingly being used in control environments where the cost of system failure is very much greater than the cost of the system itself. [Som89] states that there are more and more safety critical systems coming into use where the human costs of a catastrophic systems failure are unacceptable. The goals of a real-time design methodology should include provision for a high degree of tolerence to failure: failure either in the hardware or software of the system. Few methodologies for real-time systems, including that described in the previous chapters, consider reliability from their early stages. The resilience of the system to failure is often considered an implementation issue and treated as a 'characteristic' of the system that can be 'bolted' on top of a design at a later stage. This belief is one that lies at the heart of many system failures and is fundamentally unsound.[3].

This section of the evaluation presents a very brief overview of the concepts of hardware and software resilience against failure. We then discuss how the new real-time design methodology and supporting execution platform can provide resilience.

6.4.1 Overview of Reliability Issues

In this section we present a very brief overview of the subject of system reliability. For a more indepth survey see [AL81] and [BRT78]. [BRT78] defines the reliability of a system as

[3]Personal Communication with Ken Jackson

> a measure of the success with which the system conforms to some authoritative specification of its behaviour

Deviation from this behaviour is caused by faults in both hardware and software. There are four generally recognised kinds of fault that can occur in a computer system [BW89]. These are:

1. Inadequate specification. [Lev86] states that the majority of faults in complex systems stem from inadequate specifications.

2. Faults introduced in translating the specification into a design.

3. Faults caused by interference in the communications subsystem.

4. Faults caused by the failure of processors at runtime.

Our methodology does not concern itself with the first source of faults; we assume that the designer is presented with a correct and proven system specification. The work of [Mul79], [Mai86], [LB], [Alf77] and [Hen80] amongst many others, consider the problems of generating complete and unambiguous requirements specification documents from which the system design can be made.

One of the aims of structured design methodologies is to present a set of steps for transforming the specification of a system into an equivalent design. If the specification is proved to be correct, then the set of steps guarantees that the system design does not violate certain invariant properties of the specification. This ensures that the number of faults arising during the translation of specification to design is kept to a minimum. Ideally, the set of steps should be mechanised. Our methodology presents a set of well defined steps that start with a correct specification of the requirements and generates an implementation of the system in a distributed environment. There is however, a problem with the methodology. It does not assume that the requirements specification is written in any particular form. The methodology assumes that the environmental triggers, the data transformations and the ARCs can be abstracted from this specification.

In some safety critical real-time systems, faults introduced during the translation from the specification to the design are minimised through the techniques known as N-version programming [CA78]. In this, the real-time processes are designed in several ways. At run-time the results from each different implementation of a process are compared to

generate a majority concensus result. This approach is very expensive in terms of development costs.[4]

The third source of errors; those caused by problems with the communications network in the distributed implementation of the system are typically dealt with by provision of a layered communications protocol [Tan81], [Hal88], [GMK88]. It is typically the responsibility of the data link layer in the ISO OSI Reference Model to provide an error free communications medium. However, in the static analysis of the timing properties of the real-time

tasks, we should be aware that there is a definite probability of failure of communications messages. This failure probability should be incorporated

into the overhead costs of propagating updates to remo te copies of replicated data.

The fourth source of errors in a real-time system are those caused by processor failure. There are generally two methods to circumvent the problems caused by processor failure. These are dynamic and static redundancy. In dynamic redundancy, on detection of a failure, a copy of the appropriate real-time processes and data entities are moved to another processor. This solution does not fit with the static framework presented within the real-time design methodology. We would have no way of knowing the schedulability of the new set of tasks on the system after the reconfiguration.

Static redundancy is more preferable when considering the deterministic aspects of the system after failure; the solution also fits well with our real-time design methodology. With our model of a real-time system, static redundancy means that we replicate the real-time transactions and the data entities. The replicas are placed on separate processors. The static analysis of the timing aspects of the design then includes the overhead of these redundant transactions in the worst case scenarios for the processors.

One set of transactions in the system is known as the primary set. The other, identical set is known as the redundant set. On triggering of a task, the transaction scheduler sends the control tokens to the transactions in the primary transaction set. On detecting a failure, the transaction scheduler simply changes its transaction lists to show

[4]N-version programming also introduces significant overheads at run-time as the results from the different implementations of a process need to be compared before a final result is generated. The technique could be applied to our real-time design methodology provided this overhead is considered in the static analysis of the deterministic properties of the system

that some of the redundant transactions should now be used in place of those primary transactions on the failed processor. Any currently active transactions on the failed processor are aborted by the transaction scheduler. The redundant, replicated data entities are considered just like those copies of the data entity used by a completely separate task. When the primary task has completed a critical region, the redundant copies are updated at the same time as other copies of the data entity. In this way the redundant task is kept upto date with its primary task.

An important question arises and that is how the transaction scheduler knows a processor has failed. A standard way of achieving this is to have a 'heartbeat' message propagated throughout the network. Each node in the network can have a periodic task dedicated to propagating the heartbeat to the other nodes. The overhead of this task must be considered in the static timing analysis.

In addition to replicating the transactions and data entities, the execution environment should provide some method to replicate the scheduling mechanisms, to prevent against failure of their allocated processors.

6.5 Conclusions

[Gom84] proposes a set of requirements for a real-time systems design method. Our design method is considered against these requirements.

Dataflow Oriented

[Gom84] states that a dataflow oriented approach is appropriate for real-time systems design because the data in these systems may be considered to flow from input, through a set of software transformations, to the output. Methods such as Structured Analysis/Design, DARTS and MASCOT all exhibit dataflow characteristics. Our method is less a dataflow and more a functional approach. The designer specifies the transformations that the data must go through in response to some trigger; the methodology guides the construction of the control flow that should be imposed on the transformations. There is no concept in the methodology of a complete and explicit ordering of transformations on data until the transaction precedence diagrams are constructed. The ARCs are used to express partial orderings where necessary. These TPGs do not express the flow of data through a task. However, dataflow diagrams are used in other methodologies to aid the decomposition of the application into tasks; we have a different set of criteria for this decomposition.

Task Communication and Synchronisation

It is essential for processes in a real-time system to communicate and synchronise their actions. In our model of the real-time system, and the methodology that goes with it, there is no explicit communication at the task level. Implicit communication takes place through shared data entities but the 'sending' and 'receiving' tasks in this communication have no knowledge of the state of the task they are communicating with. Synchronisation is required to ensure the consistency of the real-time database but this is handled by the appropriate critical region schedulers and not explicitly within the task.

At the transaction level explicit synchronisation is imposed between transactions where necessary. This synchronisation is enforced through the control flow of the transaction precedence graph. Communication between transactions is again through shared data entities. In this communication though, each party to the communciation knows the state of the other.

Information Hiding

Together with decomposition, encapsulation, or information hiding, is an extremely important 'tool' in the design of real-time systems. The advantages of information hiding, are that the 'modules' in the system are self contained. This makes the system more modifiable and as a consequence more maintainable. Our methodology uses information hiding in a similar way to MASCOT. In MASCOT, all accesses to a data entity are by means of an access procedure. In our methodology, a task can only access an entity after 'consulting' the controller (critical region scheduler) associated with the entity. In all the transactions considered so far we talk of the transaction reading or writing the entity. In an implementation this reading and writing would be implemented as access procedures. These procedures are however simpler than those of MASCOT since conflicting data access is not a concern.

6.5.1 Research Objectives

The first chapter described the research objectives as primarily the study of the non-functional requirement of meeting hard real-time deadlines. It was proposed that this requirement should be considered with respect to existing real-time design methodologies to test their validity for the design of hard real-time systems. The research concluded that the existing design methodologies typically leave the achievement of task deadlines to a later stage in the design methodology. Often the objective is not attained without significant fine tuning of the design

and its implementation.

The work proposes that in order to consider the deadlines, a suitable model of execution is required. This model is the transaction. The research identified the concurrency control necessary between conflicting transactions on the same data entities as the major source of non-determinism within the model. In considering this concurrency control from an early stage in the design we can reduce its effect at run-time and so make it easier to meet real-time deadlines. The main research objectives have been met. We have considered the role of the shared data entity in the addition to the flow of control and flow of data within the system. Each presents a complementary view of the real-time system.

To support the data entity viewpoint, the research has proposed a notation and designed a simple CASE tool. This tool allows the designer to do the following:

- Create context diagrams. These illustrate the real-time computer system in the context of the environment. The sensors that trigger the computer system and those devices that the computer system can control are illustrated.
- Describe the data entities.
- Describe the characteristics of the tasks. The deadlines and minimum re-triggers times for the tasks can be specified.
- Describe the transactions for each task in terms of the actions on shared data entities.
- Generate the data entity viewpoint notation. The Data Dependency Rings can be drawn for each data entity.
- Generate the transaction precedence graphs. The graphs are automatically generated from the data entity viewpoint.
- Automatic generation of an allocation scheme. A naive allocation approach as described in Chapter 4 is implemented.
- Scheduling tests. Tests 1 and 2 as described in Chapter 6 have been implemented.

One important objective of the research is to consider how the new viewpoint and associated design methodology can be integrated with other methods. In some respects, the research has failed to do this. A

complete methodology has been described which can be used in isolation for a class of systems. Given additional time we should consider the place of the methodology in the wider field of more general methods.

6.5.2 Contribution To The State of the Art

The research described in this thesis considers an important area that is not considered other design methodologies. That area being the role of the shared data entity in the meeting of hard real-time deadlines. In considering this role, the implicit control flow imposed on otherwise independent real-time tasks can be taken into account from the early stages of the design. This enables static analysis to be carried out on the design to test for the schedulabilty of the tasks.

In addition, the research has described a hierarchical design for the construction of run-time support for real-time systems. This run-time support, in the form of the three scheduling mechanisms, is very flexible and provides advantages for building fault tolerant computer systems.

The final major contribution to the state of the art that was made during the research is the specifcation and design of the ship control system. This example is larger than typical examples described in the literature and was used to illustrate the shared entity viewpoint in designing a real-time system. The experience in using this viewpoint is described in the second appendix.

6.5.3 Final Comments

This chapter presented an evaluation of the real-time design methodology described in the preceding chapters. The method was compared with other methodologies such as MASCOT, JSD and Structured Analysis/Design. The design methodology has many advantages as well as some disadvantages over established techniques for designing systems. In an attempt to consider the run-time effects of implementation issues such as the concurrency control we have developed a design methodology that considers the system from a different viewpoint to traditional design methods. This 'data dependency viewpoint' should be complementary to the traditional control and data flow viewpoints. Consequently parts of the new methodology should be able to be used in conjunction with existing methodologies. The extent of this integration remains to be seen however.

Appendix B describes a complete example of the use of the method

from the analysis of the requirements of the Ship Control System to its implementation in a distributed database environment. The use of the method on the example has highlighted its strengths and weaknesses. The major weakness of the method is that the supporting notation, the data dependency ring, is clumsy both to construct and use. In addition, and perhaps more important, the notation attempts to capture both the data dependencies between the concurrent tasks and the control flow within a task that has been extracted from the specification. Although successfully expressing the data dependencies, the DDR notation is not really suited to expressing the control flow. Labelling the perimeter of the rings with the ARCs and selection information is clumsy, difficult to follow and results in redundant information (for example the ARCs are expressed in each DDR that has the transaction with the ARC). However, to support the notation, it is hoped that by following a functional approach to the design of the task the designer does not need to specify the complete control flow within the task, but instead just the transformations that are required of the data.

The conversion of a set of non-periodic tasks into a set of periodic tasks by considering worst case situations where each task re-triggers at the earliest possible time, appears to be a justified and powerful tool. Although the analysis to test the schedulability of the set is lengthy and involved, analysis is possible and can be automated. The test for schedulability of a task set in a hard real-time environment has not been considered in other methodologies. In our methodology, the test is central in driving the allocation of transactions and replicated data entities, to the physical processors in a distributed real-time database environment.

Finally, to conclude, the methodology considers the real-time application from a functional, data transformation driven, viewpoint where the concept of data dependencies between independent tasks is central. The methodology, although not without problems, considers aspects of hard real-time systems design glossed over by other methodologies. To carry out a full evaluation of the method, it should be tested on a real live application significantly more complex than the ship control system of Appendix B. The following chapter considers how the method may be strengthened in several directions and details desirable further work.

Chapter 7

Conclusions

Neptune, the Mystic

7.1 Overview

In many real-time computer systems the ability to deal with external events as fast as possible is not sufficient to guarantee the correct functioning of the system. For the class of *hard* real-time systems external events must be responded to within strict deadlines. Should these deadlines be missed, then the system has failed. Many real-time system design methodologies fail to recognise these strict temporal constraints on the execution of the system. Real-time system designs are often 'tweaked' by knowledgeable 'gurus' in order to achieve the required performance. This thesis attempts to provide guidance for the design of hard real-time systems by describing a step by step methodology. This methodology is applicable to applications with a large, and well defined use of a real-time database.

The introductory chapter describes the general problems of real-time systems. The second chapter describes the general characteristics of real-time database systems. These characteristics include the need for time constrained accesses; short lifetimes for the data; high update to read ratios and high availability. The failings of conventional database management systems when faced with these characteristics, are discussed. The chapter goes on to provide the motivation for using a transaction based design approach for hard real-time systems. The transaction model provides atomicity, permanence of results and recoverability: all very desirable characteristics. Finally, the chapter considers the non-deterministic aspects of the transaction model. The

concurrency control protocols necessary to protect shared data are identified as the major source of non-determinism in the model.

The next chapter described a model for an executing real-time database system. The model treats the real-time system as being constructed from a set of tasks. Each task has a single trigger and a single response. Some tasks are described as real-time tasks. These tasks have a trigger from some external event in the controlled, or monitored, environment. Each task is made up of a set of transactions. Each transaction reads a number of database entities and, optionally, updates one entity. The orderings between the transactions are partially specified by the designer in the form of application requirements constraints (ARCs). A complete ordering is found through working out the data dependencies between the transactions. The chapter also discusses the concept of a critical region on a data entity. In decomposing a task into a number of transactions, a 'functional' data transform oriented approach as opposed to an explicit control flow approach, is proposed. The control flow necessary between the transactions is worked out using the data dependencies and the ARCs.

In the fourth chapter several new notations are presented. The first, the Data Dependency Ring (DDR), presents the real-time system from the viewpoint of the data dependencies between the tasks and transactions. This notation has a ring for each data entity. The transactions that use the entity are listed around the outside of the ring, grouped in their respective tasks. The notation also captures some of the enforced orderings between transactions, the ARCs. This information expressed in this notation is used to generate the complete control flow graphs for each task. These graphs ensure that the ARCs are satisfied in addition to serialising all conflicting transactions within the task. The second notation, the transaction precedence graph (TPG), is used to present this complete control flow through the transactions of the task. The DDRs and the TPGs for the real-time system can then be used to guide an allocation of data entities and transactions to processors in a distributed, replicated real-time database environment. This chapter describes a step by step process that takes the system requirements specification and generates the DDR and TPG representations in addition to the allocation schemes.

The next chapter discussed the problem of hard real-time scheduling. Several solutions from the literature are presented. It is stated that in order to achieve a 100% success rate at meeting task deadlines an analysis of the tasks and a static placement of transactions to processors are required. Using the information generated in the TPG represen-

tations of the tasks, a static analysis of the Earliest Deadline First (EDF) scheduling policy is given. Worst cases of task triggerings are identified and the success of the scheduling policy is evaluated. This chapter also described a hierarchical scheduling environment in which the real-time tasks can execute. This hierarchy consisted of a task scheduler that recognises the triggerings of the tasks; a critical region scheduler that implements the EDF scheduling policy and controlled access to the shared data entities and the transaction scheduler that uses an implementation of an executing petri-net to control the ordering of transaction execution within a triggered task.

The sixth chapter considers the advantages and disadvantages of the methodology, static analysis and execution platform. This chapter also discusses the problems of real-time system reliability and shows how the new methodology can deal with some of these problems. A large example, the Ship Control System, is presented in an appendix. The methodology is demonstrated on this example.

7.2 Further Directions

The research work described in this thesis is not a complete solution to the problem of designing hard real-time systems. The work needs to be enhanced by further research. Some of the major directions for this research are outlined in the sections below.

7.2.1 Integration with other methods

Other design methodologies approach the system from different viewpoints. For example, MASCOT and DARTs consider the system from data flow approach whereas the methodology presented in this thesis approaches the system from the data dependencies between fundamental transformations (transactions) of the data. Each viewpoint has its merits. The dataflow approaches model the data flowing from the environment through a set of operations and back to the environment. The data dependency viewpoint can be transformed into a control flow viewpoint and is 'closer' to the implementation of the system.

It is important to understand where the different design methodologies overlap. Parts of each methodology may be used in different aspects of the systems design. The methodology presented in this paper is strong in terms of transforming a diagrammatic representation of the system into a design and eventual implementation. Other methodolo-

gies are stonger in capturing aspects of the behaviour of the application. Combining the methodologies could result in a design method that effectively captures the behaviour of the application in an intuitive form as well as providing a step by step method for automatically generating a design.

7.2.2 Moving from specification to design

The transition from a requirements specification document through to the design and implementation should be smooth and methdological. However, as pointed out in [NS90] and [KR89] the specification may require substantial reorganisation. This is due to the different concerns of the specification and design stages. [NS90] states that the specification is meant to be a complete and unambiguous description of the systems operational behaviour, whereas the system design is concerned with fitting those requirements onto a rigid and restricted host environment. Our methodology does not conform to this definition of the design stage. Instead, starting with the requirements, the methodology leads to a host environment that is suited to the requirements. Further research is required to understand exactly what is required of the specification in order to follow our methodology. Existing formal and non-formal specification techniques should be investigated to determine their suitability for use with the transaction based real-time design method.

7.2.3 Improvements to the CASE tool

The CASE tool described in Appendix A is, at present, limited in function. The current version is intended only as a demonstration of some of the aspects of the DDR notation and accompanying analysis. In addition to making changes to the human computer interface to improve the use of the tool, there are many functional additions that can be made. The main area that would benefit from improvement is in the allocation of transactions and data entities to processors and the accompanying static analysis to check for schedulability. At present, the allocation scheme does not recognise the constraint of a limited number of processors; the static analysis currently assumes complete tasks are assigned to a single processor.

7.2.4 Evaluation of design quality

There is much scope within the design methodology presented for designing different systems to solve the same real-time problem. Chapter 4 states that the decomposition of the application into sub-systems is very much dependent on the skill of the designer. In addition, the design of the real-time database and decomposing the task into a set of transactions operating on this database also depends on the skill and experience of the designer. The transaction precedence graphs generated may or may not be good representations of the task. Transaction precedence graphs are representations of programs and consequently familiar techniques for testing the structuredness of programs can be used as a measure of goodness [BJ66], [BS72].

Other parts of the design should also be tested for some measure of 'goodness'. The complexity of the data dependency rings for a particular design indicates the degree of data dependencies between the otherwise independent tasks. Some qualitative measure should be assigned to this complexity to judge the design. This will be a measure of how well the real-time database has been designed and how well the tasks have been decomposed into transactions.

Finally, there should also be some measure of the goodness of the allocation scheme of transactions and data entities to processors. The static analysis determines whether the particular allocation is effective or not, but an allocation may result in some processors being more heavily loaded than others. The measure of goodness of the allocation scheme should take into account the load placed on each of the processors.

7.3 Concluding Remarks

For complex real-time systems, it is important to consider the non-functional,hard timing constraints early on in the design of the system. Current real-time system design methodologies result in a system with correct functional requirements but achieving the desired performance is often considered later on, not always with success. Increasing use is being made of database management systems to provide a general purpose platform for the implementation of the real-time system. The work presented in this thesis considered the role of such databases in the construction of hard real-time systems. The methodology recognises the important non-functional requirement of task deadlines early on in an attempt to provide the system with the required performance from

the start. Although not without problems, it is hoped that the work presented in this thesis is a step toward achieving deterministic hard real-time database systems.

Bibliography

[AJ89] R. Agrawal and H.V. Jagadish. Recovery Algorithms for Database Machines with Non-volatile Main Memory. In *6th International Workshop on Database Machines (Eds. Boral and Faudemay)*, AT&T Bell, Labs, New Jersey, 1989. Springer-Verlag.

[AL81] T. Anderson and P.A. Lee. *Fault Tolerence Principles and Practice.* Prentice-Hall, 1981.

[Alf77] M.W. Alford. A Requirements Engineering Methodology for Real-Time Processing Requirements. *IEEE Transactions on Software Engineering*, SE-3(1):60–69, Jan. 1977.

[All81] S.T. Allworth. *Introduction to Real-Time Software Design.* McMillan, 1981.

[All83] J. Allen. Maintaining Knowledge about Temporal Events. *Communications of ACM*, 26(11):832–843, Nov. 1983.

[Ast84] K.J. Astrom. *Computer Controlled Systems.* Prentice-Hall, 1984.

[AT88] E.A. Abbadi and S. Toueg. The Group Paradigm for Concurrency Control Protocols. In *ACM SIGMOD Conference Proceedings*, Jun. 1988.

[Bat87] G. Bate. *The Official Handbook of MASCOT.* Defence Reseach Information Centre, Glasgow, 1987.

[Ben88] S. Bennet. *Real-Time Computer Control : An Introduction.* Prentice Hall, 1988.

[BG81] P.A. Bernstein and N. Goodman. Concurrency Control in Distributed Database Systems. *ACM Computing Surveys*, 13(2), June 1981.

[BJ66] C. Boehm and G. Jacopini. Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules. *Communications of ACM*, 9(5):366–371, May 1966.

[Bok81] S. Bokhari. A Shortest Tree Algorithm for Optimal Assignments Across Space and Time in a Distributed Processor System. *IEEE Transactions on Software Engineering*, SE-7(6), Nov. 1981.

[BRT78] P.A. Lee B. Randell and P.C. Treleaven. Reliability Issues in Computing System Design. *ACM Computing Surveys*, 10(2):123–65, Feb. 1978.

[BS72] J. Bruno and K. Steiglitz. The Expression of Algorithms by Charts. *Journal of the ACM*, 19(3):517–525, Jul. 1972.

[BS79] P.A. Bernstein and D.W. Shipman. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, SE-5(3):203–216, May 1979.

[BW89] A. Burns and A. Wellings. *Real-Time Systems and their Programming Languages.* Addison Wesley, 1989.

[CA78] L. Chen and A. Avizienis. N-Version Programming: A Fault Tolerence Approach to Reliability of Software Operation. In *Digest of Papers, The Eigth Annual International Conference on Fault Tolerent Computing*, 1978.

[Cam86] J.R. Cameron. An Overview of JSD. *IEEE Transactions on Software Engineering*, SE-12(2):222–240, Feb. 1986.

[Cla89] J. Clarke. Sema group plc, new malden, surrey. personal communication, 1989.

[DeM78] T. DeMarco. *Structured Analysis and System Specification.* Yourdan Press, 1978.

[Dix87] K. Dixon. *Benchmark Times for the Relational Processor : Report No. DVME785/TN6.* Ferranti, Nov. 1987.

[Dix88a] K. Dixon. *Design Specification of the Relational Processor : Report No. DVME785/DS.* Ferranti, May 1988.

[Dix88b] K. Dixon. *The Ferranti DVME 785 Relational Processor : Report No. 6902, Issue 5.* Ferranti, Sept.1988.

[Eic89] M.H. Eich. Main Memory Database Research Directions. In *6th International Workshop on Database Machines (Eds. Boral and Faudemay)*. Springer-Verlag, 1989.

[ELT82] R. Ma E.Y.S. Lee and M. Tsuchiya. A Task Allocation Model For Distributd Computing Systems. *IEEE Transactions on Computers*, C-31(1), Jan. 1982.

[Fab74] R.S. Fabry. Capability-based Addressing. *Communications of the ACM*, 17(7):403–411, Jul. 1974.

[GMK88] H. Garcia-Molina and B. Kogan. Achieving High Availability in Distributed Databases. *IEEE Transactions on Software Engineering*, SE-14(7):886–896, Jul. 1988.

[Gom84] H. Gomaa. A Software Design Method for Real-Time Systems. *Communications of ACM*, 27(9):938–949, Sep. 1984.

[Gom86] H. Gomaa. Software Development of Real-Time Systems. *Communications of ACM*, 29(7):657–668, Jul. 1986.

[Gra78] J.N. Gray. Notes on Database Operating Systems. In *Operating Systems - An Advanced Course (Eds. Bayer, Graham and Seegmuller)*, pages 393–481. Springer-Verlag, 1978.

[GS78] T. Gonzalez and S. Sahni. Algorithms for Scheduling Independent Tasks. *Journal of the ACM*, 25(1), Jan. 1978.

[Hal88] F. Halsall. *Data Communication, Computer Networks and OSI: 2nd Edition.* Addison-Wesley, 1988.

[Har80] D. Harel. On Folk Theorems. *Communications of the ACM*, 23(7):379–389, Jul. 1980.

[Har87] D. Harel. *Algorithmics : The Spirit of Computing.* Addison-Wesley, 1987.

[Hen80] K.L Heninger. Specifying Software Requirements for Complex Systems: New Techniques and their Application. *IEEE Transactions on Software Engineering*, SE-6(1):2–13, Jan. 1980.

[HHT89] X. Tan H. Hong and D. Towsley. A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System. *IEEE Transaction on Computers*, 38(12):1736–1744, Dec. 1989.

[Hoa85] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, 1985.

[HP88] D.J Hatley and I.A. Pirbhai. *Strategies for Real-Time System Specification.* Dorset House Publishing, 1988.

[HZ] M. Hamilton and S. Zeldin. Higher Order Software - A Methodology for Defining Software. *IEEE Transactions on Software Engineering*, SE-2(3).

[Jac83] M. Jackson. *System Development.* Prentice Hall, 1983.

[Jac84] K. Jackson. Introduction to basic MASCOT principles. *IEE Colloquium Digest*, 113, Dec. 1984.

[JBW86] M. Drabrowski J. Blazewicz and J. Weglarz. Scheduling Multiprocessor Tasks to Minimize Schedule Length. *IEEE Transactions on Computers*, C-35(5), May 1986.

[JSC85] K. Ramamritham J.A. Stankovic and S.C. Cheng. Evaluation of a Flexible Task Scheduling Algorithm for Distributed Hard Real-Time Systems. *IEEE Transactions on Computers*, C-34(12), Dec. 1985.

[KET76] R.A. Lorie K.P. Eswaran, J.N. Gray and I.L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications of ACM*, 19(11):624–633, Nov. 1976.

[KN84] H. Kasahara and S. Narita. Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing. *IEEE Transactions on Computers*, C-33(11), Nov. 1984.

[KR81] H.T. Kung and J.T Robinson. On Optimistic Methods for Concurrency Control. *ACM Transactions on Database Systems*, 6(2):213–226, 1981.

[KR89] D. Kalinsky and J. Ready. Distinctions Between Requirements Specification and Design of Real-Time Systems. In *Proceedings of the Second International Conference on Software Engineering for Real-Time Systems.* IEE, Sep. 1989.

[LA90] S-T. Levi and A. Agrawala. *Real Time System Design.* McGraw Hill, 1990.

[Lam78] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of ACM*, 21(7):558–656, Jul. 1978.

[Lam81] B. Lampson. Atomic Transactions. In *Distributed Systems - Architecture and Implementation (Eds. Goos and Hartmanis)*, pages 246–265. Springer-Verlag, 1981.

[Law88] J. Lawton. *An Assessment of the DIOMEDES Distributed Database Product.* SEMA Group, 1988.

[LB] B.H. Liskov and V. Berzins. An Appraisal of Program Specifications. In *Software Specification Techniques (Ed. Gehani and McGettrick)*, Massachusetts Institute of Technology. Addison Wesley.

[LC85] B.H. Liebowitz and J.H. Carson. *Multiprocessor Systems for Real-Time Applications.* Prentice-Hall, 1985.

[Lei80] D.W. Leinbaugh. Guaranteed Response Times in a Hard Real-Time Environment. *IEEE Transactions on Software Engineering*, SE-6(1), Jan. 1980.

[Leu89] J. Y-T. Leung. A New Algorithm for Scheduling Periodic Real-Time Tasks. *Algorithmica*, (4):209–217, 1989.

[Lev86] N.G. Leveson. Software Safety: why, what and how. *ACM Computing Surveys*, 18(2):125–63, Feb. 1986.

[Lis85] B. Liskov. The Argus Language and System. In *Distributed Systems Methods and Tools for Specifications, An Advanced Course (Eds. Paul and Siegert).* Spinger-Verlag, 1985.

[LL73] C.L. Liu and J.W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, 20(1), Jan. 1973.

[LM88] P.D. Lawrence and K. Mauch. *Real-Time Microcomputer System Design: An Introduction.* McGraw-Hill, 1988.

[LY86] D.W. Leinbaugh and M.R. Yamini. Guaranteed Response Times in a Hard Real-Time Environment. *IEEE Transactions on Software Engineering*, SE-12(12), Dec. 1986.

[Ma84] R.P-Y. Ma. A Model to Solve Timing Critical Application Problems in Distributed Computer Systems. *IEEE Computer*, 1984.

[Mai86] T.S.E. Maibaum. A Logic for Formal Requirements Specification of Real-Time Embedded Systems, 1986.

[Man67] G.K. Manacher. Production and Stabilization of Real-Time Task Schedules. *Journal of ACM*, 14(3):439–465, Jul. 1967.

[Mar82] C. Martel. Preemptive Scheduling of real-time task on multiprocessor systems. *Journal of the ACM*, 29(3), Mar. 1982.

[MC70] R.R. Muntz and E.G. Coffman. Preemptive Scheduling of Real-Time Tasks on Multiprocessor Systems. *Journal of the ACM*, 17(2), Apr. 1970.

[Men79] D.A. Menasce. Locking and Deadlock Detection in Distributed Databases. *IEEE Transactions on Software Engineering*, SE-5(3), May 1979.

[Moo68] J.M. Moore. An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Management Science*, 15(1):102–109, Mar. 1968.

[MP84] S.M. McMenamin and J.F. Palmer. *Essential Systems Analysis.* Yourdan Press, 1984.

[MSS82] P. M. Melliar-Smith and R.L. Schwartz. Formal Specification and Mechanical Verification of SIFT: A Fault Tolerant Flight Control System. *IEEE Transactions on Computers*, C-31(7):616–630, Jul. 1982.

[Mul79] G.P. Mullery. CORE - A Method for Controlled Requirements Specification. In *Proc. 4th International Conference on Software Engineering.* IEEE Computer Society Press, 1979.

[Mul89] S. Mullender. *Distributed Systems.* Addison Wesley/ACM Press, 1989.

[Mur] T. Murata. Modeling and Analysis of Concurrent Systems. In *Handbook of Software Engineering.* Van Nostrand Reinhold Company.

[NS90] M. Nejad-Sattery. An Extended Data Flow Diagram Notation for Specification of Real-Time Systems, PhD Thesis, 1990.

[Pap79] C.H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of ACM*, 26(4):631–653, Oct. 1979.

[Par72] D.L. Parnas. On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of ACM*, 12(12):1053–1058, Dec. 1972.

[PBG87] V. Hadzilacos P.A. Bernstein and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison Wesley, 1987.

[Pet77] J.L. Peterson. Petri Nets. *Computing Surveys*, 9(3):223–252, Sep. 1977.

[PS89] J.L. Peterson and A. Silberschatz. *Operating System Concepts.* Addison Wesley, 1989.

[RS84] K. Ramamritham and J.A. Stankovic. Dynamic Task Allocation in Hard Real-Time Distributed Systems. *IEEE Software*, Jul. 1984.

[Sah76] S.J. Sahni. Algorithms for Scheduling Independent Tasks. *Journal of the ACM*, 23(1):116–127, Jan. 1976.

[Sal74] J.H. Saltzer. Protection and the Control of Information Sharing in Multics. *Communications of the ACM*, 17(7), 1974.

[SCS88] K. Ramamritham S.C. Cheng and J.A. Stankovic. Scheduling Algorithms for Hard Real-Time Systems - A Brief Survey. *IEEE Computer*, 21, 1988.

[Sha90] A. Shackleton. Sema group plc, new malden, surrey. personal communication, 1990.

[Sho83] M.L Shooman. *Software Engineering: Design, Reliability and Management.* McGraw-Hill, 1983.

[Sle91] P.M. Sleat. Real Time Databases. *SEMA Group Technical Journal*, Feb. 1991.

[Som89] I. Sommerville. *Software Engineering.* Addison Wesley, 1989.

[Spe89] A.Z. Spector. Distributed Transaction Processing Facilities. In *Distributed Systems (Ed. S. Mullender).* Addison Wesley/ACM Press, 1989.

[SR89] J.A. Stankovic and K. Ramamritham. The Spring Kernal : A New Paradigm for Real-Time Operating Systems. *ACM Operating Systems Review*, 23(3), Jul. 1989.

[Sta88] J.A. Stankovic. Misconceptions About Real-Time Computing: A Serious Problem for Next-Generation Systems. *IEEE Computer*, 21(10):10–19, Oct. 1988.

[Sut88] A. Sutcliffe. *Jackson System Development.* Prentice Hall, 1988.

[Tan81] A.S. Tanenbaum. *Computer Networks.* Prentice-Hall, 1981.

[Tay89] D.S. Taylor. The SUCCESSOR Infrastructure for SMCS, SEMA GroupReport No. SMCS/INF/000044/3A, Feb. 1989.

[TIM87] T. Kameda T. Ibaraki and T. Minoura. Serialisability with Constraints. *ACM Transactions on Database Systems*, 12(3):429–452, Sep. 1987.

[Tow86] D. Towsley. Allocating Programs Containing Branches and Loops within a Multiple Processor System. *IEEE Transactions on Software Engineering*, SE-12(10), Oct. 1986.

[WCE80] M.T. Lan W.W. Chu, L.J. Holloway and K. Efe. Task Allocation in Distributed Data Processing. *IEEE Computer*, 13(11), Nov. 1980.

[WKSG89] J-Y. Chung W-K. Shih, J.W.S. Lui and D.W. Gillies. Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error. *ACM Operating System Review*, 23(3):14–28, Jul. 1989.

[WM86] P.T Ward and S.J. Mellor. *Structured Development for Real-Time Systems, volume 1,2 and 3.* Yourdan Press, New Jersey, 1986.

[Wol87] O. Wolfson. The Overhead of Locking (and commit) Protocols in Distributed Databases. *ACM Transactions on Database Systems*, 12(3), Sept. 1987.

[WZS87] K. Ramamritham W. Zhao and J.A. Stankovic. Scheduling Tasks with Resource Requirements in Hard Real-Time Systems. *IEEE Transactions on Software Engineering*, SE-13(5):564–577, May 1987.

[YC78] E. Yourdan and L. Constantine. *Structured Design.* Yourdan Press, 1978.

[Zav82] P. Zave. An Operational Approach to Requirements Specification for Embedded Systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–269, Mar. 1982.

Appendix A

An Interactive CASE Tool

This appendix describes an interactive CASE tool that has been written to aid the design and implementation of real-time systems. The tool described is the GEM (Digital Research) version. A version for X-Windows is being developed, and apart from minor differences in the 'front end', will be much the same.

A.1 Introduction To Methodology

It is assumed that the reader is familiar with the real-time design methodology on which this CASE tool is based. A brief overview is be given here.

A real-time system consists of a set of independent tasks. Each task has a separate and distringuishable triggering event in the environment being monitored or controlled. A task is the set of actions, or 'transactions' necessary to respond to this event. The transactions manipulate a set of data entities. Theoretically, more than one transaction may access a data entity at the same time. If these concurrent transactions have conflicting requirements on the data entity, they are sequentialised. This prevents conflicting concurrent access. Most non-real-time database systems perform this sequentialisation at run-time. This however leads to transactions whose start times are delayed by other conflicting transactions. This delay is not known and so analysis of the timing aspects of the tasks is difficult.

The designer of the real-time system should not have to worry about the need to sequentialise conflicting concurrent transactions. He should be able to concentrate on the transformations to the data entities that are necessary to respond to an environment event. This CASE tool

is designed to work out the complete flow of control necessary within the transactions of a task. This flow of control is worked out as the transactions are designed. When the design is finished, a complete flow of control is calculated for the transactions within a task. The sequentialisation of transactions necessary to preserve the state of the database is generated by the tool prior to run-time.

Since the effect of other transactions on a given transaction within the same task has been determined, the worst case execution times for the task can be determined. This enables static analysis to be performed to determine whether a given task will meet its hard real-time deadlines. This analysis is carried out by the tool. In addition, the tool can suggest an allocation of transactions and data entities such that good use of concurrency is made.

A.2 The WIMP Environment

The CASE tool has been written with a 'user friendly' front end based on Windows, Icons, Mice and Pointers (WIMP). The CASE tool environment consists of

- Drop Down Menus for selecting actions
- Forms for entering data
- Windows in which information and results are displayed

These are now described.

A.2.1 Drop Down Menus

Drop down menus are the means by which the user selects actions in the CASE tool. The main menu for the CASE tool is displayed on the very top row of the screen. The mouse is used to move the pointer onto one of the submenus. A drop down submenu now appears as in figure A.1. The mouse can again be used to select the suboption by moving the pointer up and down. If, at any time, the mouse moves out of the drop down menu, the menu disappears. As the pointer is moved down the submenu, the options are highlighted. In figure A.1, the 'save' option is highlighted. Pressing the left hand mouse button selects the highlighted option.

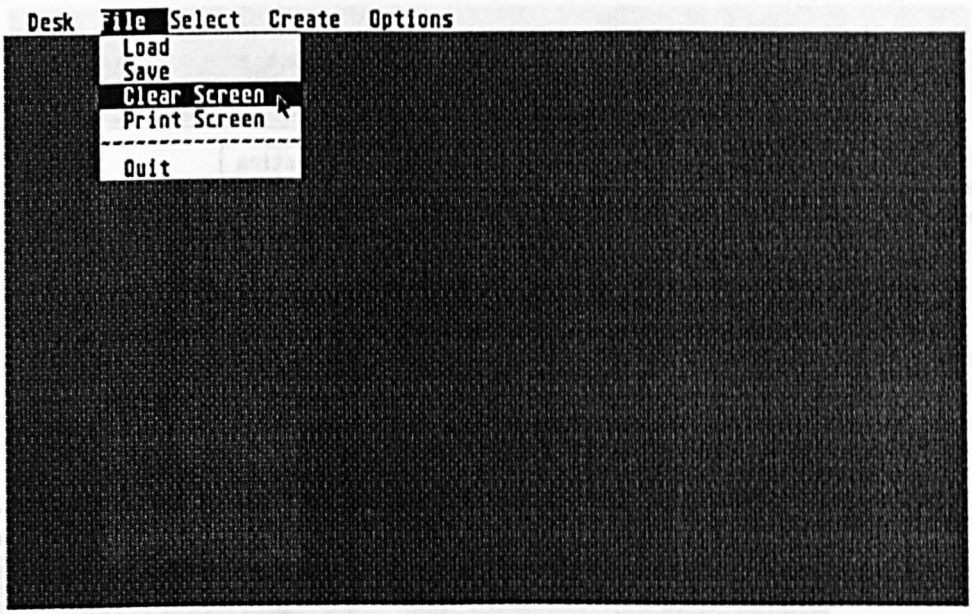

Figure A.1: Selecting Options from a Submenu

A.2.2 Forms

The form is the means by which the user presents information to the CASE tool. An example form is shown in figure A.2. On initial display of a form, the cursor is found in the first field. Any text or numbers that are typed are entered into this field. The cursor can be moved in a number of ways. These are :-

1. The **TAB** key. Pressing the TAB key moves the cursor onto the next field. If the cursor was initially on the last field then pressing TAB causes it to 'wrap-around' to the begining.

2. The **ARROW** keys. The left and right arrow, or cursor control, keys move the cursor within the current field. The up and down arrow keys move the cursor to the previous or next field respectively.

3. The **MOUSE**. The mouse can be used to move the screen pointer to any field. If the left hand mouse button is pressed while pointing at a field, the cursor will move to that field.

When data has been enetered into all the fields, the user may then either select the *DONE* or *CANCEL* options by pointing to the appropriate

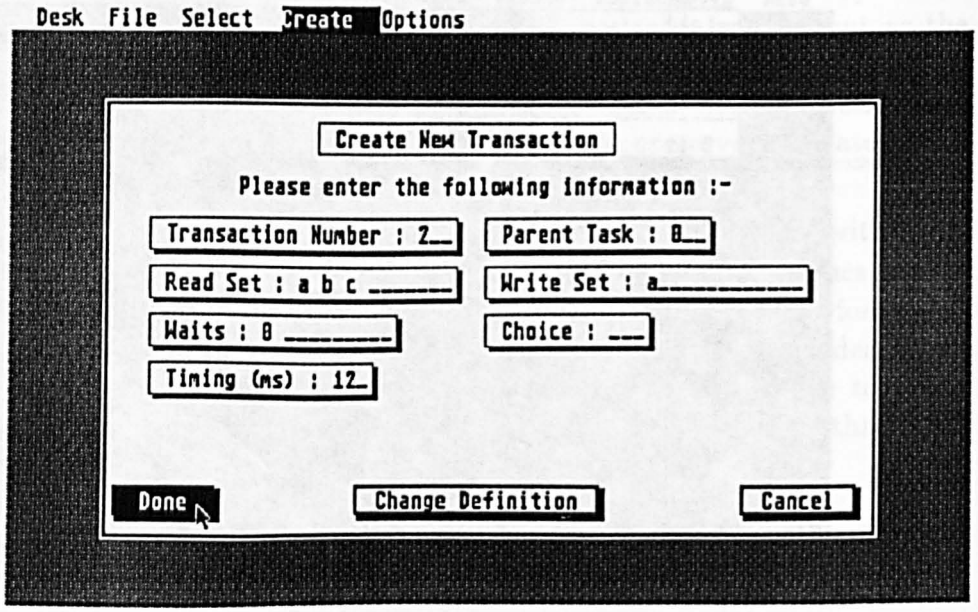

Figure A.2: An Example Data Entry Form

button and pressing the left hand mouse button. Pressing *DONE* makes the CASE tool act on the new data that has been entered into the form. Pressing *CANCEL* make the CASE tool ignore the information that has been entered.

The fields of a given form are initially set to blanks on starting the CASE tool. From then on, any information entered into the form is 'remembered' between succesive displays of the form.

A.2.3 Windows

The window is the means by which the CASE tool displays information to the user. There are three types of window used in the CASE tool.

1. Message windows
2. Permanent display windows
3. Temporary display windows

The message windows display simple information and typically make the CASE tool wait for a simple response from the user to indicate that the message has been read. For example, the initial title screen for the

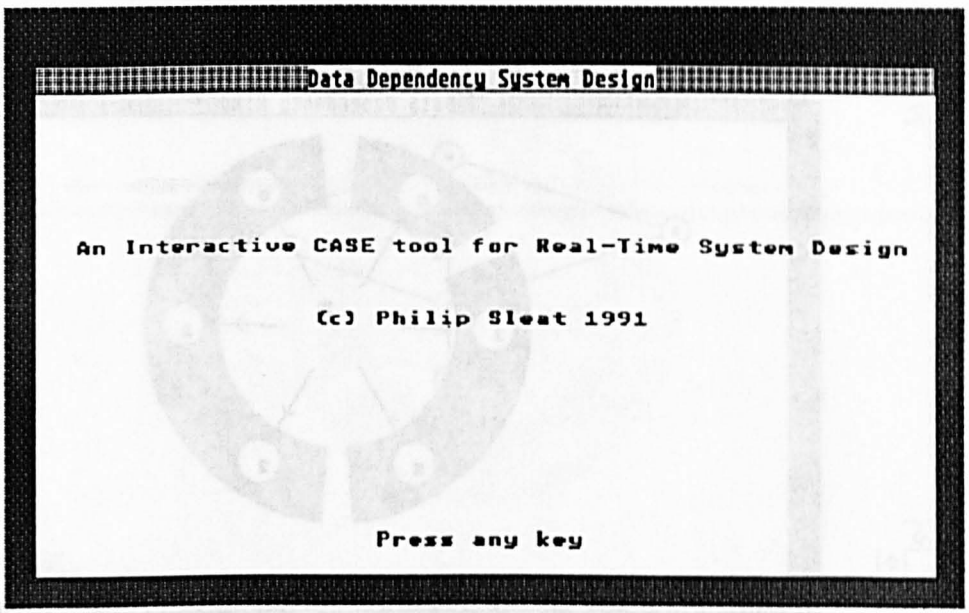

Figure A.3: Initial Title Screen in a Message Window

CASE tool shown in figure A.3 is an example of a message window. The user must press any key before the window is removed and the CASE tool continues. A further type of message window is the 'Alert box'. This represents an internal error or an error in the users input data. The alert box is removed by pointing to the *OK* button and pressing the left hand mouse button.

The permanent display windows show information all the time they are 'active'. The information that is displayed is typically only one screens worth. An example of such a window is the DDR window shown in figure A.4. This is made active by chosing the *Select* then *DDR* options from the menu. All the time a permanent window is displayed, the drop down menus and data entry forms can still be used. The window is removed by selecting the *File* then *Clear Screen* options from the menu.

Temporary display windows display are for the display of more than one screen's worth of information. The only temporary display window used by the CASE tool is for the display of the transaction precedence graph of a task. An example of such a display window is shown in figure A.5. The temporary display window has 'scroll arrows' on the right hand and bottom sides. Pointing to one of these arrows and pressing the left hand mouse button causes the temporary display window to scroll in

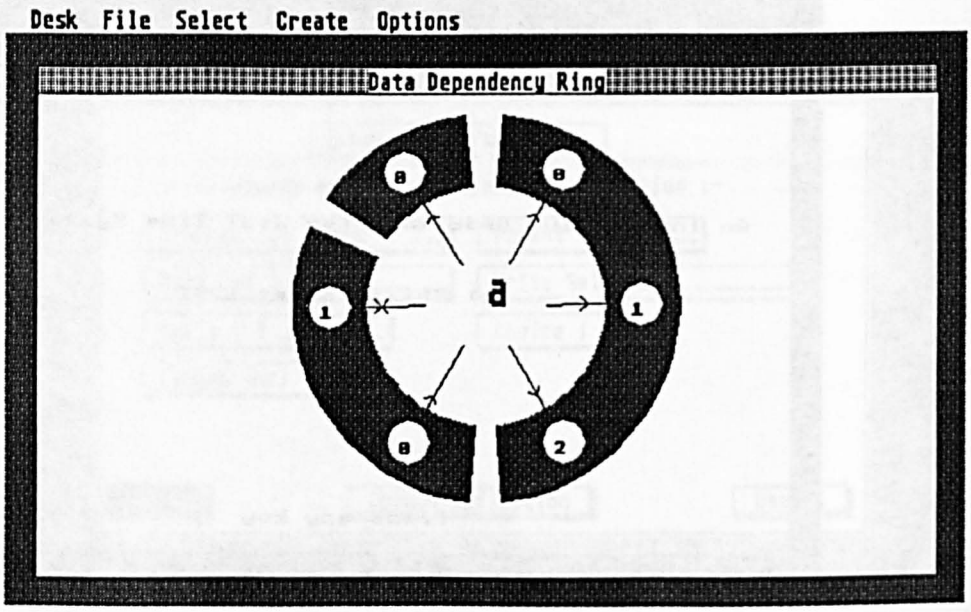

Figure A.4: DDRs displayed in a Permanent Window

the appropriate direction. The temporary display window also has a 'close' button in the top left hand corner. Clicking on this removes the temporary display window. All the time the temporary display window is active, the main menu and data entry forms are inactive.

A.3 Creating A Real-Time System

On initial starting, there is no real-time system defined in the CASE tool. To start a new design, a name for the system is needed. This name is entered by selecting the *Create* then *RTS* options from the main menu and then entering upto eight characters.

A.3.1 Entering the System Devices

Each real-time system will have a number of external devices and actuators through which information about the controlled environment is gathered and through which control is fed back to the environment. These external devices can be represented in the real-time system design by selecting the *Create* and then *External Device* options from the main menu. A form is displayed in which to enter the characteristics

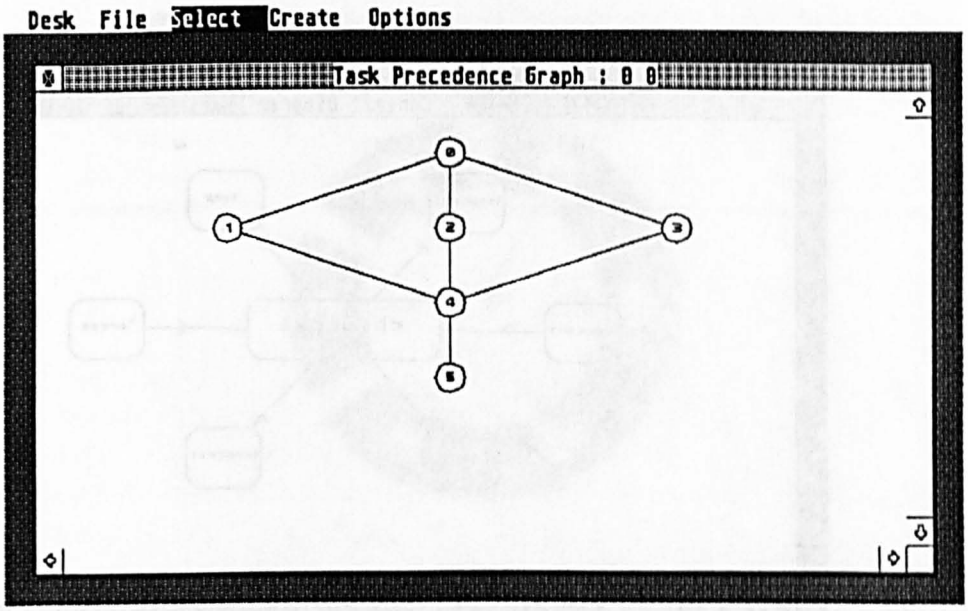

Figure A.5: Transaction Precedence Graph in a Temporary Window

of the device. The following information is required :-

- *Name.* Enter the name of the external device (upto eight characters).
- *Stimulus or Response.* If the external device is an input device i.e. the device sends the computer system information about the environment then enter 'S' for stimulus. If the device is an actuator for the computer system to control the environment, enter an 'R' for response.
- *Period.* For periodic stimuli enter the period of triggering of the device. If this field is left blank and the device type is still a stimulus then it is assumed that the stimulus triggering is sporadic. This field should be left blank for 'response' device types.

A.3.2 Displaying the Real-Time System

The context diagram for the real-time system can be displayed by chosing the *select* and then *RTS* options from the menu bar. The context diagram is displayed in a permanent information window. The real-time computer system is represented in the centre of the diagram. The

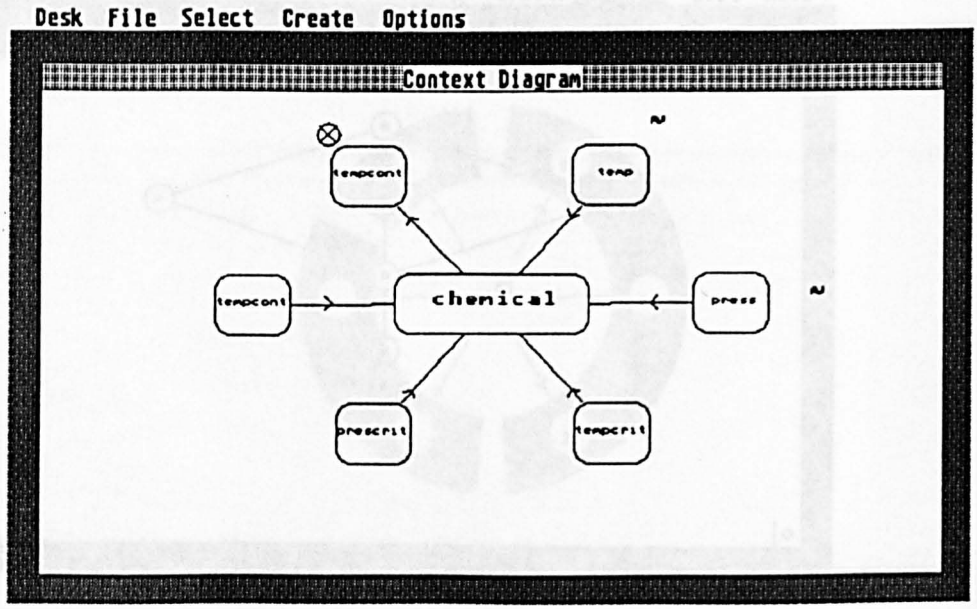

Figure A.6: An Example Context Diagram

external devices linked to the real-time system are represented around this. Figure A.6 shows an example context diagram. Periodic tasks are signified with a '~' placed next to their name. Response devices are signified with an actuator symbol (circle with a cross) placed next to their name.

Since the context diagram is displayed in a permanent information window, further devices may be added by selecting the *Create* and *External Device* options from the menu bar. After each additional external device is specified, the context diagram is redrawn.

A.4 Creating a Data Entity

A real-time database system consists of a number of data entities and a set of tasks that transform these entities appropriately. In designing a real-time system the major data entities that are used must be identified. In selecting the *Create* and *DDR* a name can be given to each of the data entities used. There is one Data Dependency Ring for each data entity used in the system. In naming the rings, the entities are named in the real-time database.

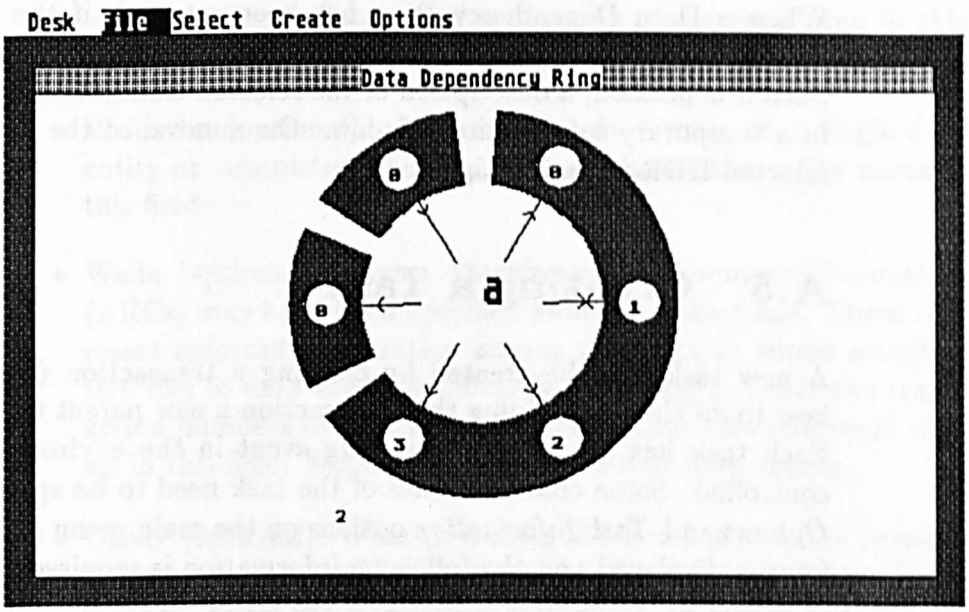

Figure A.7: An Example DDR

A.4.1 The Data Entity Viewpoint

The actions of the real-time system may be displayed from the viewpoint of the role that each data entity plays in the system. By choosing *Select* and then *DDR* from the menu the Data Dependency Ring, for a named entity is displayed. The name of the entity must be entered. Should the name not be known a '?' may be entered in the name field. This will display a list of all the known data entities. The correct name can then be entered.

The Data Dependency Ring is displayed in a permanent information window. An example is shown in figure A.7. The data entity name is shown in the center of the ring. Around the outside, the tasks and transactions that use the ring are listed. For transactions that write to the entity, there is an arrow pointing to the centre of the ring. For transactions that read the entity, the arrow points to the edge of the ring. Enforced 'waits' within transactions are listed around the outside of the ring. In the example, transaction 3 of task 1 must always wait for transaction 2 of task 1. Since the data dependency rings are displayed in a permanent information window, new transactions can be added at any time (see later for how to do this) and the ring is redrawn after each new transaction is added.

When a Data Dependency Ring has been selected, if the pointer is moved within one of the transaction circles and the left hand mouse button is pressed, a description of the selected transaction is displayed in a temporary information window. On removal of the window, the selected DDR is again displayed.

A.5 Creating a Task

A new task may be created by creating a transaction (see later for how to do this) and giving that transaction a new parent task number. Each task has a separate triggering event in the environment being controlled. Some characteristics of the task need to be specified. The *Options* and *Task Information* options on the main menu allow this. A form is displayed and the following information is required :-

- Task Number.
- Deadline. The real-time deadline of the task must be specified in milliseconds relative to the triggering time of the task.
- The Minimum Re-Trigger Time. This is the shortest interval between successive re-triggerings of the same task. For periodic tasks this corresponds to the period. For non-periodic tasks, the MRT typically corresponds to some physical characteristic of the environment being controlled and the triggering device.

A.6 Creating a Transaction

A transaction may be created by chosing the *Create* and the *Transaction* options from the main menu. A form is displayed for entering descriptive information about the transaction. The following information is required :-

- Transaction Number (mandatory). Transactions have numbers starting at 0. They are simply used as a means to distinguish between transactions and do not imply any serialisation between successive transactions.
- Parent Task Number (mandatory). Each transaction is contained within a parent task. Enter the task number in this field. For a new task, enter the next available task number.

- Read Set (optional). A transaction may read from data entities already defined. Enter the list of entity names read from in this field, separating each entity name by a space.

- Write Set (optional). A transaction may write to a single data entity on completion of its processing. Enter the entity name in this field.

- Waits (optional). Some Application Requirement Constraints (ARCs) may have been specified among transactions. These represent enforced serialisation among transactions where serialisation due to data access conflicts does not exist. Enter the transaction numbers of all those transaction that this one must wait for in this field. Separate multiple numbers with spaces.

- Choice (optional). Where a transaction is one of a choice between several, one transaction will be the parent from which the choice is made. Enter the transaction number of the parent in this field.

- Timing (mandatory). Enter the worst case execution time for the transaction in this field. Timings are in milliseconds.

A.7 Task View of the Real-Time System

A control flow viewpoint of each task may be displayed. Choose the *Select* and then *Task* options from the main menu and then enter the required task number. The control flow viewpoint of the task is automatically generated so that the data dependencies among transactions and ARCs are met. A transaction precedence graph is displayed in a temporary information window. Where an arc is drawn from a higher (nearer the top of the screen) transaction to a lower transacton, there exists either an ARC between these transactions or else the transactions conflict and must be serialised. Where there is no such arc, the transactions do not conflict and may execute concurrently. An example transaction precedence graph is shown in figure A.5.

Some transaction precedence graphs are too large to be displayed on the screen. The information window that displays the TPG has scroll buttons on the bottom and right hand sides. Pointing to these and pressing the left hand mouse button causes the window to scroll in the appropriate direction displaying more of the TPG. The current view on the TPG is displayed as a coordinate in the title of the information window. The top left page of the TPG is initially displayed and given

the coordinate (0,0). Scrolling to the right increases the first index. Scrolling down increases the second index.

A.8 Allocation Schemes

A.9 Static Temporal Analysis

Given that the temporal characteristics of the tasks have been defined a static analysis of these characteristics is possible. Given the execution times of each of the component transactions, the worst case execution time for the complete task can be determined. Given this, together with the deadline and the minimum re-trigger times for a task, the allocation scheme generated using the allocation scheme option can be tested for conformance.

To carry out this static analysis, choose *Options* and then *Scheduling advice* from the main menu. A data entry form is then displayed. Enter the level of analysis (1,2 or 3) required and select the done button to start the analysis.

A.9.1 Analysis Level 1

At scheduling analysis level 1, the task set is tested to check that it can be serialised though shared resources such that there is enough processing time available to complete all tasks while at the same time ignoring the effects of conflicting access to shared data. If the task set fails with this simple analysis then the set cannot be scheduled in an environment where the conflicting access to shared data is respected.

The analysis is carried out by considering the minimum retrigger times for the tasks (these are entered by using the Options then Task information choices from the menu). A worst case is constructed where each task using the shared resource is triggered repeatedly at its re-trigger time. The time period upto the point where all tasks simultanteously re-trigger is then considered. This point is calculated by incrementing a counter in steps of the largest of the re-trigger times. At each step, the count is divided by each of the other re-trigger times. If all divide exactly into the count then the consideration point (T_c) is found. The number of triggerings of each task upto this point is then found and the total execution time of all these task triggerings calculated. If this total time is greater than the consideration interval T_c then there is

not enough processor time for all the task triggerings and the static analysis shows that the task set is not sound. If the total time is less than or equal to the consideration time T_c then the analysis should continue with levels 2 and 3 to check that the tasks can be serialised through resources and still meet deadlines.

A.9.2 Analysis Level 2

At scheduling level 2, the task set is tested to check that tasks can execute in a non-preemptive environment. Once the task has started executing it executes through to completion. A worst case scenario is generated for each task. This scenario assumes that each task with sooner deadlines are triggered at the same time as the task in question. These must all execute before the task in question. The time that these tasks complete is calculated; this is the earliest time that the task being considered may start executing. Using this time, whether the task completes before the deadline is checked.

A.9.3 Analysis Level 3

At scheduling level 3, the task set is tested in a preemptable environment. Each task may be interrupted once by tasks with an earlier deadline. A worst case scenario is generated for each task. In this worst case scenario the task in question is prevented from execution by the current execution of the longest task with an earlier deadline. The soonest time that the task in question may start is then calculated. In addition, the task is preempted by triggerings of tasks after the task in question has started. These tasks have sooner deadlines. The slack of the task in question must then accommodate the intial delay plus the partial execution of the task and also the execution of those tasks with sooner deadlines.

For scheduling the three scheduling levels, the tool performs the check and reports on the validity of the task set for the level of scheduling chosen.

A.10 Saving, Loading and Printing

The current real-time system may be saved to disk using the *File* and *Save* options from the main menu. On selecting this option, a file selector box is displayed. The user may move around the file system

to select the directory in which the real-time system is to be saved. Enter the name under which the real-time system is to be saved. An extension of '.RTS' will be appended to the name and the real-time system saved.

To load an existing real-time system from disk choose the *File* and *Load* options from the main menu. On selecting this option, a file selector box is again presented. This is used to choose the real-time system to be loaded. On loading a real-time system, any previous information entered is overwritten.

A limited printing facility is provided. Chose the *Print* option from the main menu. A file selector box is displayed. On chosing an appropriate name, the current screen will be saved to this named file on the disk. The screen is saved in Degas PI3 format and may be loaded and printed by the Degas Art Package. In addition, a tool exists to convert from Degas PI3 format into PostScript format.

A.10.1 File Formats

The real-time system is saved to disk as a simple ASCII file describing the characteristics of the system. An example real-time system is described by the following file.

```
RTS chemical
Device temp                    010 S
Device press                   005 S
Device tempcrit                000 S
Device prescrit                000 S
Device tempcont                000 S
Device tempcont                000 R
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
Device                         000 N
```

```
ENTITY 00 a
ENTITY 01 b
TASK 00 000 000 010
TRANS 00 RNNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 01 BNNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 02 RRNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 03 RRNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 04 RRNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 05 RRNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00

TRANS 06 RWNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 07 RNNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 08 NBNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 09 NBNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
TRANS 10 WRNNNNNNNNNNNNNNNNN
      000 000 00 00 00 00 00
              00 00 00 00 00
END
```

The file consists of a set of five record types, each record being constructed from fixed position fields. The records are as follows :-

1. RTS followed by the name of the real-time system.
2. Device followed by the name of the device, the period and then the type of the device (Stimulus (S), Response (R) or not used

(N)).

3. **ENTITY** followed by the entity number and then the entity name.

4. **TASK** followed by the task number, minimum re-trigger time and deadline and then the number of transactions in the task.

5. **TRANS** followed by the transaction number and then a flag for each entity in the system representing the transaction's use of the entity (Write (W), Read (R), Read and Write (B), no use (N)). On the second line of the transaction record there is the worst case execution time for the transaction, the transaction which choses this one and the transactions that this one must wait for due to ARCs. On the final line of the transaction record are the transactions that this one optionally choses on completion.

6. **END** represents the end of the real-time system file.

In the transaction records, where a transaction refers to some other transaction the transaction index is incremented by one. A '00' in the field represents an index of -1. For example if the transaction has 01 02 03 00 00 in the waits field this means that the transaction waits for transactions 0, 1 and 2 to complete. The 00 and 00 at the end of the wait list represent -1 i.e. a wait for nothing.

A.11 CASE Tool Implementation Details

The CASE tool has been written in 'C' and compiled using the Prospero C compiler for the Atari ST personal computer. The front end to the tool makes much use of the Graphical Environment Manager (GEM) interface.

The source code is divided into nine 'C' files and two 'C' include files. These files are as follows :-

- **MAIN.C** This holds the calls to initilise the application, start the main program loop, and close down the application on completion.

- **CASE.C** This holds the main program loop and various other routines such as the loading and saving of real-time systems code. The drawing of the context diagrams is handled within this file.

- **DDRVIEW.C** This draws the real-time system from the viewpoint of the chosen data entity.
- **TASKVIEW.C** This draws the real-time system from the viewpoint of the control flow through a task.
- **ALLOC.C** This generates a suggested allocation scheme assigning transactions and data entities to processors.
- **SCHEDULE.C** This analyses the task set to ensure schedulability.
- **GRAPHICS.C** This handles all the routines to draw lines, circles, filled pie-slices etc. This file needs to be changed if the tool is to be ported to another windowing system.
- **GEMSTUFF.C** This file handles the GEM oriented code such as displaying and getting the information from, data entry forms and windows. This file needs to be changed if the tool is to be ported to another windowing system.
- **UTILITY.C** This file contains several utilities.
- *CASE.H* This is the header file that describes the structures of internal storage. In addition, various constants are defined here such as the maximum number of data entities or transactions a real-time system can have. Change these figures and recompile to increase the capacity of the tool.
- *CASE3.H* This header file defines names and associated GEM numbers for each of the menu option, data entry forms and the fields within these forms.

APPENDIX A. AN INTERACTIVE CASE TOOL

Appendix B

A Ship Control System

Whereas simple examples given in the literature such as the bottling plant application can demonstrate aspects of the data dependency design methodology, a much larger application is needed to demonstrate the complete methodology. This appendix describes a large and complex real-time application. The treatment of the application using the data dependency design methodology is not intended to be accurate and complete. This appendix is intended to demonstrate the stages that are undertaken to generate a final design. In any real design and implementation, further work would be required to accurately match the design to the requirements. The would involve a great deal of consultation with the 'customer' to ensure that what is provided is what is wanted. Consequently, the design process would typically follow an iterative path. This iteration is not evident from the relatively simple process described in this chapter.

To test the effectiveness of the methodology, the example should have a large shared database content. This appendix shows the stages from the study of the initial outline specification of the requirements to a complete design for a distributed real-time database system. The appendix is organised into four main sections. First, an overview of the application is given. This section corresponds to a vague, and probably incomplete, statement of the requirements of the application. In the second section the database requirements of the application are described. In addition, this section identifies the real-time tasks and describes them in terms of actions on the real-time database. In the third section, a data dependency ring analysis is carried out and the transaction precedence graph for each real-time task is constructed. The final section describes an allocation of transactions and database entities to processors in a distributed network and tests for the schedulability of

this configuration.

B.1 The Ship Control System - Requirements

Modern C3 systems typically have requirements for large and sophisticated databases [1]. This example considers a fleet of commercial ships which is to be fitted with an embedded computer control system. This control system is responsible for automatically monitoring and controlling the state of the ships engines; guiding the ship between destinations; accepting new courses and commands from the operator; monitoring and sending communications between the ships of the fleet. These functions are now briefly elaborated on.

B.1.1 Overview of the ships function

The operating conditions of the engines in each ship need to be carefully controlled. Each ship has two engines. The fuel consumption of each engine is to be monitored; should the fuel levels in the storage tanks drop below a certain threshold, then some remedial action is required. The best form of action should be that the ship is automatically guided to the nearest fleet refueling tanker. If there is not enough fuel to reach this, the ship should stop and wait for a refueling tanker to arrive. Each engine has strict controls on the environment in which it can operate. The temperature of the engines must be monitored. A coolant can be introduced into the engines to maintain a constant temperature. Should the temperature reach a critical state before the coolant can take effect, then the engines should be shut down. The engines speed in revolutions per minute must be carefully considered. Should the speed go beyond a threshold, the engines are working too hard and must be shut down. In normal operating conditions, the speed of the engines is determined by the speed of the ship, in knots, required by the crew together with external influences such as headlong winds etc. The engines must obviously work harder to maintain the same overall speed when the ship is steered into high winds. In emergency conditions, to slow the ships down, the engines can be quickly switched into reverse

[1] It has been estimated that the control system for the British Navy's Type-23 Frigate will spend 90% of all processing time carrying out database functions [Cla89].

2.

The control system is responsible for guiding the ship between successive destinations specified in a course plan set out by the crew. At regular intervals, the control system should read the ship's current bearing from a bearing device. This current bearing should be compared with the required bearing based on the set course. The position of the ship's rudders is adjusted to ensure that the ship maintains a true course to the required location. The control system can also be operated in manual mode where the commands entered on the operator's console directly control the ships rudder.

The control system is responsible for accepting commands from the crew via a sophisticated graphics display console. The control system also displays various aspects of the behaviour of the ship, such as the speed and direction of the vessel, on the display console.

Each ship is only one part of a large fleet of ships. It is important that these ships maintain not only voice communications via radio (which is outside of the scope of the computer control system) but also data communications which are handled by the computer control system. Each ship can send one of a standard set of messages to any other ship in the fleet: each ship has a unique address and a packet radio system is used to send the messages. At regular intervals, the control system will send a current location message to the fleet controller - a central computer system at the fleet headquarters. Other sorts of messages that can be sent include orders for fuel (directed to the fueling tankers) and mayday messages indicating that the ship is in distress.

The control system is responsible for interpreting the data from a set of radar and sonar devices. The radar is typically used to guide the ship in adverse weather conditions, so as to avoid collisions, by adjusting the ship's course if obstacles are detected. The sonar device is used in a similar way to avoid the problem of 'bottoming-out' in shallow waters. Should it be detected that an unavoidable collision is about to occur i.e. the ship does not have time to change course effectively, then the ship should be stopped under the control of the computer system and a set of mayday messages automatically sent out to other fleet ships. Should the control system detect a mayday message then it is fleet policy that the ship should be guided to the site of the distressed vessel if it is within a certain distance.

[2]Apparently, at normal 'cruising speed' the British Navy's new Type-23 Frigate can switch the direction of the engine and stop within its own length. At higher speeds, the engines cannot be switch so easily and the stopping distance is increased [Sha90]

The control system has a natural decomposition into five subsystems. These subsystem match the five overviews just given. The subsystems are

1. Engine Control Subsystem
2. Ship Guidance Subsystem
3. Operator Subsystem
4. Communication Subsystem
5. Collision Detection (Radar/Sonar) Subsystem

B.1.2 The Physical Environment

This section describes the 'environment' that the control system operates in. This environment is defined in terms of the physical devices that can send data from the external world to the control system and those devices that can be used to change the outside world. These devices are shown in an extended Context Diagram in figure B.1. Those devices marked with an 'actuator' symbol (circle with a cross) are the output control devices. Those devices marked with a tilde are devices that periodically supply the control system with information from the environment. Those devices marked with a 'C' are input devices that will return information about the controlled environment when 'Consulted' by the control system. The devices with no markings are input devices that automatically trigger the control system when they have new information ready to send.

Each device in the external environment generally relates to a specific subsystem. The subsystem decomposition diagram for the control system is shown in figure B.2. In addition to these devices, there is a real-time clock that generates an interrupt at regular intervals and that is used for controlling some of the periodic functions such as the reading of the engine temperatures and the redisplaying of the operators display console as well as recording event times in a ships log.

B.1.3 The Real-Time Triggers and Tasks

After having identified the external devices we must now describe the actions or tasks that must be under taken associated with each device. These descriptions are organised according to the subsystem in which

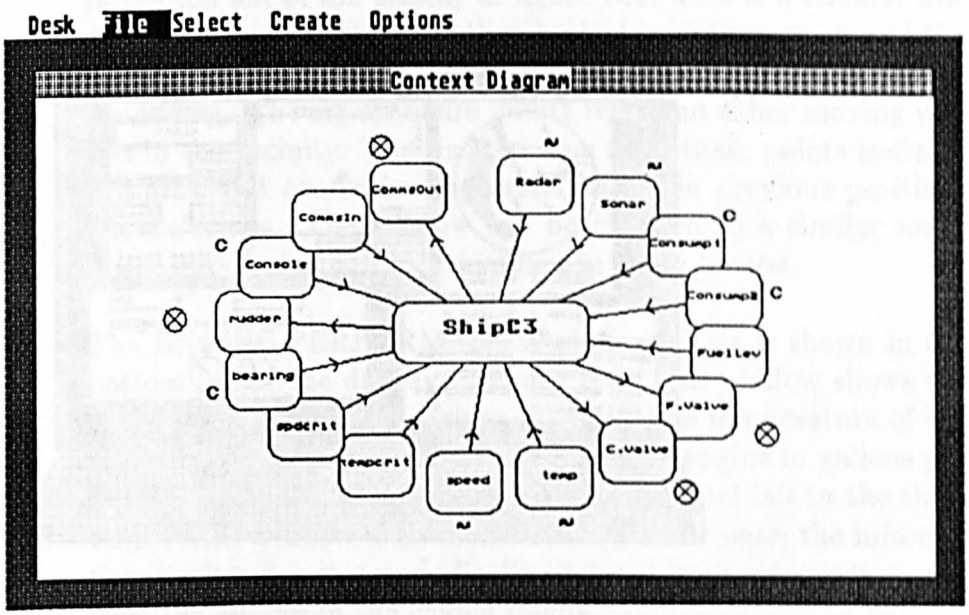

Figure B.1: Extended Context Diagram for the Ship Control System

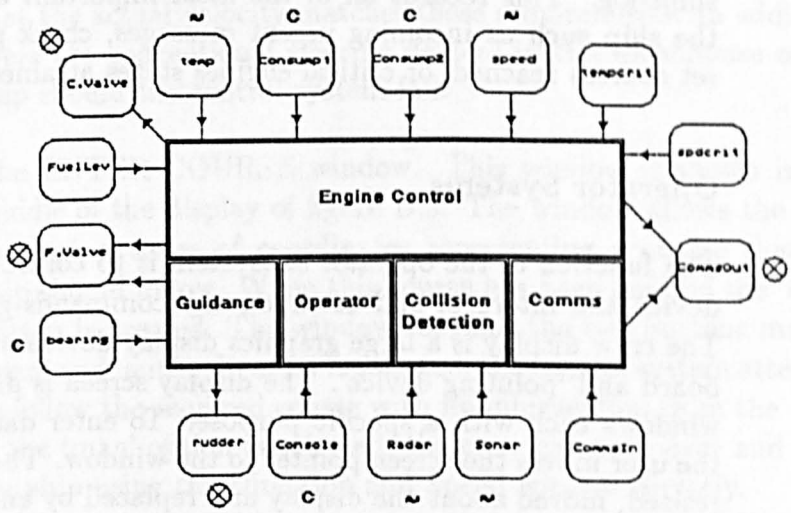

Figure B.2: Subsystem Decomposition Diagram for the Ship Control System

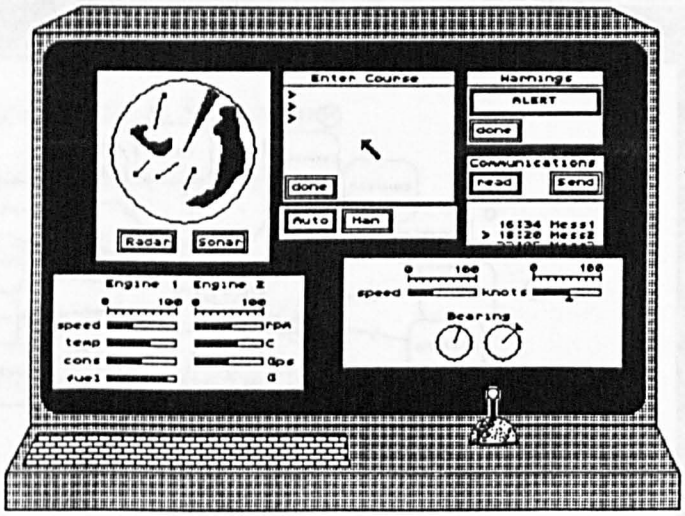

Figure B.3: The Crew Display Terminal

the associated task appears. In addition to the actions described in the follow sections, most of the real-time tasks write an entry to the ships-log. This records all of the most important actions concerning the ship such as incoming urgent messages, check point locations on set courses reached, or critical engines states attained.

Operator Systems

The function of the operator subsystem is to control the crew display device and interpret and act upon any commands given by the crew. The crew display is a large graphics display device with a built in keyboard and 'pointing device'. The display screen is divided into several windows each with a specific purpose. To enter data into a window, the user moves the screen pointer to the window. The windows may be resized, moved about the display and replaced by an appropriate icon. The crew display is shown in figure B.3.

The functions of the crew display can be described by defining the actions of the component windows of the display. All actions to be taken as a result of commands entered at the crew display should be executed as soon as possible. The functions of the crew display windows are now defined.

- The OBSERVATIONS window. The observations window is shown in the top left of the display in figure B.3. This is a circular display that shows the state of the physical environment around the ship. In the figure, the window is showing the results of the radar processing. The small square points represent other moving vessels in the vacinity. The lines coming from these points indicate the predicted course for the vessel based on previous positions and velocities. The window can be changed to a similar sonar image and back again by selecting the sonar button.

- The ENGINE PERFORMANCE window. This is shown in the bottom left of the display in figure B.3. The window shows the speed of the ship's two engines (in rpm); the temperature of the engines in °C and the consumption of each engine in gallons per second. In addition, the window shows the fuel left in the ships tanks. This window is an information window only; the information displayed in it is periodically updated from information read from the devices in the engine rooms.

- The SHIP DIRECTION/SPEED window. This window is shown in the bottom right of the display. This window displays the actual bearing (in degrees) and speed (in knots) of the ship. In addition, there are 'sliders' such that the crew may enter the required speed and bearing. The control system automatically ensures that the actual velocity matches these requirements. In addition, there are conventional steering devices in the wheelhouse of the ship should the control system fail.

- The ENTER COURSE window. This window is shown in the middle of the display of figure B.3. The window allows the crew to enter a series of coordinates representing a course that the ship should follow. When this course has been entered the 'done' button is pressed. The window contains the two buttons marked 'man' and 'auto'. If 'auto' is selected, the control system attempts to follow the required course with no intervention from the crew. If the 'man' option is selected, the crew can take over and steer the ship using the Direction and Speed window directly.

- The WARNINGS window. This is shown on the top right of the display of figure B.3. This window displays any important warning message from the computer system. The message is 'flashed' in the window with an audible bell. The crew can perform no other action on the display until the 'done' button is pressed.

Such warnings are that the ship is on a collision path with some object or that the engine speed is too high to be safe.

- The COMMUNCATIONS window. This is shown in the middle right of the display in figure B.3. The window holds a scrollable list of incoming messages. A simple view enables the operator to scroll the list and inspect any message. In addition, the window has a send mode; a message may be selected from a list of standard messages using the keypad and sent to a remote ship with which data communications is possible.

In terms of real-time tasks, the operator subsystem has a separate task in the windows of the display device for each action that the user can perform . This does not include actions such as resizing the windows; these are assumed to be a function of the window management system and is not considered in this work. The dispay device is 'refreshed' at regular intervals based on the system clock. The 'input buffer' of the display device, which records the users actions, is also read at regular intervals based on the system clock.

Engine Control Subsystem

The engine control and monitoring subsystem is the largest part of the ship control system. The engines must be run at their most efficient and within strict safety constraints. This part of the control system ensures that these constraints are satisfied. The engine control system is constructed from a number of independent real-time tasks. These tasks are defined as follows.

- Critical Fuel Level. When the level of fuel in the tanks drops below a certain threshold, this task is triggered. A warning must be displayed on the crew display to show the lack of fuel. In addition, a message should be send to the nearest re-fueling tanker requesting supplies. If there is sufficient fuel left, then a course should be set for this tanker and the control system put into 'manual control' mode; the crew can then enter 'auto' mode, if required, to steer to the tanker. If there is insufficient fuel to get there, then the ship should be stopped.

- Critical Engine Temperature. If the engine temperature has reached a critical value then the amount of coolant entering the engines should be increased to bring the temperature down and the engines should be halted by cutting the flow of fuel to them. A

warning message should be displayed on the crew display terminal.

- Critical Speed. The speed of the engine has reached a critical value. The speed should be reduced by cutting the flow of fuel to them. A warning message should be displayed on the crew display terminal.
- Engine 1 consumption. The fuel consumption in gallons per second is monitored at regular intervals.
- Engine 2 consumption. The fuel consumption in gallons per second is monitored at regular intervals.
- Engine temperatures. The engine temperatures are sensed at regular intervals. The amount of coolant that enters the engine is altered, if necessary, to keep the temperature at a constant value.
- Engine speeds. The crew specify the required speed of the ship in knots. At regular intervals, the current speed of the engines in RPM is read together with the current speed of the vessel in knots. New engine speeds are calculated to match the required ship speed in knots with the actual speed in knots. This exercise is carried out at regular intervals.

Ship Guidance Subsystem

The ship guidance system is only used if the crew have selected the automatic pilot mode from the display console. The main function of the ship guidance subsystem is to steer the ship automatically through a set course of bearings until the ultimate destination is reached.

The ship guidance subsystem is invoked at regular intervals. If autopilot is not selected, then the guidance task records the ship's current location. If auto pilot is selected then the subsystem must compare the current position of the ship (found by consulting the bearing device) with the required bearing that is in the course set by the crew. The appropriate adjustments are made to the ship's rudder to ensure that the course to the required bearing remains true. If the required bearing is reached, then the guidance subsystem must steer the ship to the next point on the set course until the final destination has been reached.

The ship guidance subsystem can be switched off at any time by the operator selecting 'manual control' on the display console. If the guidance subsystem is just switched off, then the ship continues with the

current rudder and engine speed settings. A new course may only be entered into the ship guidance subsystem when the ship is in 'manual' mode.

Communications Subsystem

The function of the communciations subsystem is to handle all electronic message flow between the ship and other ships in the same fleet. The communications medium used is a packet radio network. A hardware radio device can detect a start packet. This device automatically 'screens' out those packets that are not intended for this ship; each ship has a unique identifier embedded in the start packet, which the communications device is aware of. When a start packet intended for this ship is detected, the communications device triggers the communciations subsystem which then logs the message intended for this ship. The message is then added to the message list in the communications window of the display device. The messages can have an associated priority. If the priority of an incoming message is high then a warning message is also shown on the display device. No further action can then be taken until this message is read.

All outgoing mail is handled when the display device is regularly read for new commands. If some outgoing mail has been entered, this is sent directly to the communications device which forms the message into packets. The communications device is then responsible for sending, and if necessary resending, the message. No further action is needed by the computer control system after the outgoing message has been forwarded to the communications device.

Collision Detection Subsystem

The collision detection subsystem is probably the most important part of the control system in terms of safety. The primary role of the subsystem is to ensure that the controlled ship does not collide with any other seaborne vessel and also that the ship can navigate difficult waters without 'bottoming out'. The collision detection subsystem uses both a radar and sonar device; the subsystem consists of two real-time tasks, one to handle the information from each of the devices.

The radar task is triggered when the radar device has a new set of information to be transferred to the control system. The first function of the radar task is to look for urgent collisions based on this new data. The immediate position of the ship is compared with the new set of

points representing the positions of other vessels (and land masses). If an unavoidable collision path is detected, the ship must be stopped. Given that no unavoidable collisions are detected, the radar task must further analyse the new data. The task maintains a track table that records the movement of all vessels around the ship. In addition there is an old track table that records previous locations of other vessels. The radar task must save the positions in the current track table in the old track table. The new set of radar points is then matched to vessels and the new track table constructed. If any vessel that was previously in radar site is now no longer, then its entries are deleted from the old track table. The radar task now examines each entry in the current and old track tables to predict the future positions of the vessels. If any vessel is predicted to be on a collision course with the controlled ship, then a warning message is inserted into the message list for viewing in the communications window. The radar task has strict timing constraints. Not only is it a safety critical task but also the task must be executed completely before the next set of input data has arrived to prevent the build up of extensive data queues.

The role of the sonar task is similar to that of the radar. The track table consists of both surface and underwater objects that are of interest. The sonar task must still check for unavoidable collisions, update the two track tables and check for potential collisions. Although a safety critical task like the radar task, the sonar task has a longer deadline due to the nature of the hardware and the time it takes for the sonar device to prepare each set of data.

B.2 Database Design

It has been decided to implement the ship control system using a real-time database. This database is organised as a set of tables accessible by any task that needs them. The transaction is proposed as an execution model to provide a fault tolerent execution environment. The structure and organisation of the individual tables is not of interest at this stage of the design. The individual tables are treated as 'lockable' entities. Improved concurrency may be attained through dividing the tables further, although this is not discussed in this appendix. We also do not need to know the physical location of a table at this stage, or indeed, whether a table is replicated or not.

B.2.1 The Database

This section of the appendix describes, in overview, each of the main database tables.

1. CDT (Crew Display Table). This table holds the current view of the crew display windows. The windows, input fields and state of the buttons on the crew display are represented within the database. The CDT is further divided into separate sections, or sub-tables. These separate sections may be treated as database entities in their own right; any number may be updated at the same time. By accessing the CDT, a process can treat all these separate entities as a whole. The separate entities making up the CDT are :-

   - CDT.OBS the observations window holding the radar/sonar image.
   - CDT.ENG the engine performance window table.
   - CDT.DIR the ship direction/speed window table.
   - CDT.CRS the ship course window table.
   - CDT.WRN the warnings window table.
   - CDT.COM the communications window table.

2. CTT (Current Track Table). The current track table holds information about the most recent positions of all objects within radar and sonar range of the ship. The CTT records such information as the most recent position, relative size, relative velocity, predicted future positions etc of the object.

3. OST (Operational State Table). The OST holds information about the current execution of the ship control system such as the state of the guidance system (manual or automatic) and whether the operator requires radar or sonar images etc.

4. OTT (Old Track Table). The old track table holds the previous locations and times of the objects within radar and sonar range of the ship. If one of these objects moves out of range, then the associated entries in the CTT and OTT are deleted. Before the CTT is updated, the entries in it are added to the OTT; a trace of the movements of the objects is kept. A maximum of ten entries for each object are retained in the OTT. This should be enough to work out the future positions of the objects.

5. SLT (Ship Location Table). The SLT holds details of the current and past locations of the ship. In addition, the current ship speed and location are recorded. Past locations are added to this table at regular intervals and retained for some maximum time.

6. SST (Ship Strategy Table). The SST holds the required course that the ship should follow if on automatic pilot. One part of the table holds the next location that should be reached, together with its bearing from the current location. The rest of the table holds the future locations. As the next location is reached, it is replaced by the top of the future positions list from the second part of the SST.

7. CWT (Collision Warnings Table). The CWT holds details of all those objects that are on a potential collision path with the ship. This table is used to display these objects in a different way in the observations window of the crew display, thus highlighting those objects on a potential collision path.

8. CCT (Collision Critical Table). This table holds details of those objects that are definitely within the path of the ship. If no corrective action is taken, then a collision will occur. The records in the CCT are used to alert the crew of impending collision.

9. EST (Engine State Table). The engine state table holds the required state of the the ships engines. This state is represented as the speed of the engines in rpm; the position of the rudder; the direction of the engines and the required speed of the ship in knots.

10. EET (Engine Environment Table). The engine environment table records the state of the engine environment. The table holds such information as the current fuel flows; fuel levels; position of the rudder; engine speeds and engine temperatures.

11. CIL (Communications Input Log). As each new message intended for this ship is detected, it is added to the CIL. When a message is read by the operator, it is deleted.

12. SRL (Ship Running Log). The ship's log records all significant events in the lifetime of the ship. Significant events include dangerous conditions in the ships engines such as extreme temperatures; high priority messages arriving at the ship; set locations on the course being passed through and critical collision situations.

In addition, in order to provide a uniform interface for the transactions, some hardware devices in the system are treated as database entities. This is similar to the treatment of I/O devices as special files in operating systems. The 'special' database entities, all prefixed with an 's' are :-

1. sRUD (Rudder). Writing to the rudder device changes the position of the rudder. sRUD is an output device only.

2. sDIB (Display Input Buffer). All data entered by the operator at the display device is stored in the input buffer of the device. This may be examined by reading the sDIB. This is an input device only.

3. sDSB (Display Screen Buffer). The screen image on the display device is altered by sending a copy of the CDT to the sDSB. The display device takes the information in this buffer and converts it to the screen image.

4. sCOM (Communications Device). The communications device is both an input and an output device. If new messages have arrived, the sCOM interrupts the control system; a read from the sCOM then retrieves the packet of information intended for this ship. Writing a message to sCOM makes the communications device split the information into packets and transmits it.

5. sES1 (Engine1 Speed). A read from this device returns the speed of the first engine. This is an input device only.

6. sES2 (Engine2 Speed). A read from this device returns the speed of the second engine. This is an input device only.

7. sTMP (Engine Temperature). A read from this device returns the temperature of the engine room. This is an input device only.

8. sCN1 (Engine1 Consumption). A read from this input only device returns the current fuel consumption of the first engine.

9. sCN2 (Engine2 Consumption). A read from this input only device returns the current fuel consumption of the second engine.

10. sSON (Sonar device). The sonar device causes an interrupt when it has built up the next sonar image of the environment. A read from this device then transfers the image to the control system.

11. sRAD (Radar device). The radar device causes an interrupt when it has built up the next radar image of the environment. A read from this device transfers the image to the control system.

12. sBER (Bearing device). A read from this input only device returns the current bearing and physical location of the ship.

13. sFVL (Fuel Valve). A write to this output only device changes the state of the valve introducing fuel to the engines. The engines can be slowed down or speeded up by changing the state of this valve.

14. sCVL (Coolant Valve). A write to this output only device changes the state of the valve introducing coolant to the engines. The engines can be cooled further by opening the valve.

15. sCLK (System Clock). The current time may be found by reading the system clock device.

The data entity description of the design process is now complete. To summarise, the real-time database contains twelve table entities and fifteen special entities.

B.2.2 Tasks and Database Actions

Now that the database entities have been decided upon, we are in a position to define the real-time tasks in terms of actions on the database. This section defines the tasks as sets of transactions on the database. Where some partial ordering among transactions is required, this is specified as an Application Requirements Constraint (ARC). In addition, the temporal requirements for the tasks are defined.

The timing considerations in these tasks are often arbitrary. In some cases, such as the operator tasks, the re-trigger times and deadlines are artificially small. The actual re-trigger times where there is an operator involved are likely to be of the order of seconds rather than the millisecond deadlines of other tasks. In the transaction sets for the tasks a read of a data entity is shown as the entity name prefixed with a 'r'. Similarly, a write is shown as 'w' and the name of the entity. Where a choice must be made within a set, those subsets that must be executed on each outcome of the choice are represented as indented sets of transactions.

In order to present the method for static analysis of the execution times for the tasks, we assign fixed times to the elements of a transaction. We

assume that reads and writes of a database entity take 2 and 4 units of time respectively. There is also the assumption that a read to a special device entity takes 4 units of time and a write takes 6. In a real system, these figures would be worked out for each specific entity.

The actual execution time of the transaction is not considered in the analysis. The execution time is considered insignificant compared with the reading and writing time of the entities. In a real system, these execution times would not be insignificant and must be considered in the execution time of each transaction. All timings are expressed in some arbitrary unit of time, the Time Unit, TU.

Task 1 : Read operator display device

Trigger Period 20 TU
Deadline 20 TU **Processing Set**

1. rsDIB wCDT - get the input data and make the appropriate changes to the CDT (E=6)
2. rCDT.DIR wEST - get the required engine parameters and put in the engine state table (E=6) (ARC 1)

Task 2 : Refresh operator display device

Trigger Period 20 TU
Deadline 20 TU
Processing Set

1. rEET wCDT.ENG - write the engine performance to the screen (E=6)
2. rOST rCTT rOTT wCDT.OBS - write the radar or sonar image to the screen (E=10)
3. rCDT wsDSB - write the CDT table to the device buffer (E=8) (ARC 1,2)

Task 3 : Sonar to Radar Display

Trigger Aperiodic - Minimum re-trigger time 40 TU

Deadline 20 TU Processing Set

1. wOST - Change the OST entry to indicate radar (E=4)

Task 4 : Radar to Sonar Display

Trigger Aperiodic - Minimum re-trigger time 40 TU
Deadline 20 TU Processing Set

1. wOST - Change the OST entry to indicate sonar (E=4)

Task 5 : Manual to Autopilot

Trigger Aperiodic - Minimum re-trigger time 40 TU
Deadline 20 TU Processing Set

1. rSST - check the strategy table for a course (E=2)
2. If no strategy wCDT.WRN - write a warning message (E=4)
3. rsCLK wSRL - update the log (E=8)
4. If strategy exists wOST - Show that autopilot is now operative (E=4)
5. rsCLK wSRL - update the log (E=8)

Task 6 : Autopilot to Manual

Trigger Aperiodic - Minimum re-trigger time 40 TU
Deadline 20 TU Processing Set

1. wOST - Show that manual control is now operative (E=4)
2. rsCLK wSRL - update the log (E=8)

Task 7 : Change Required Speed

Trigger Aperiodic - Minimum re-trigger time 20 TU
Deadline 20 TU **Processing Set**

1. rCDT.DIR wEST - update the Engine state table from the DIR window (E=6)

Task 8 : Change Required Bearing

Trigger Aperiodic - Minimum re-trigger time 20 TU
Deadline 20 TU **Processing Set**

1. rCDT.DIR wEST - update the Engine state table from the DIR window (E=6)

Task 9 : Respond to Warning Message

Trigger Aperiodic - Minimum re-trigger time 20 TU
Deadline 20 TU **Processing Set**

1. wCDT.WRN - clear the message away (E=4)
2. rsCLK wSRL - update the log (E=8)

Task 10 : Accept New Course

Trigger Aperiodic - Minimum re-trigger time 20 TU
No Deadline Processing Set

1. rOST (E=2)
2. If in manual mode rCDT.CRS wSST - get the new course from the window and put in the strategy table (E=6)
3. rsCLK wSRL - update the log (E=8)

Task 11 : Send Message

Trigger Aperiodic - Minimum re-trigger time 20 TU
Deadline 20 TU **Processing Set**

1. rCDT.COM wsCOM - pass the message to the comms device (E=8)
2. rsCLK wSRL - update the log (E=8)

Task 12 : Display Incomming Message

Trigger Aperiodic - Minimum re-trigger time 20 TU
Deadline 20 TU **Processing Set**

1. rCIL wCDT.COM - get the message from the communication input log and display in the communications window (E=6)

Task 13 : Fuel Level Critical

Trigger Aperiodic - Minimum re-trigger time 2000 TU
Deadline 2000 TU **Processing Set**

1. rsCLK wSRL - update the log (E=8)
2. wOST - move to manual (E=4)
3. rSLT rCTT - determine how far away the nearest fuel tanker is (E=4) (ARC 2)
4. If the tanker is within range rCTT rSLT wSST - set course for the nearest refueling tanker (E=8)
5. rSST wsCOM - send a message to the tanker to prepare for refueling at a given location in the strategy table (E=8) (ARC 4)
6. If too far away wsFVL - stop the engines (E=6)
7. wEST (E=4)
8. wCDT.DIR (E=4)

9. rSLT wsCOM - send a location message to the tanker (E=8) (ARC 6,7,8)

10. rSLT wsCOM - send a location message to fleet hq (E=8)

11. wCDT.WRN - Write out the warning message (E=4) (ARC 1,5,9,10)

Task 14 : Critical Engine Speed

Trigger Aperiodic - Minimum re-trigger time 5000 TU
Deadline 2000 TU **Processing Set**

1. wsFVL - close the fuel valves (E=6)

2. wEST - zero the speed in the engine state table (E=4)

3. wCDT.DIR - zero the speed in the display device window (E=4)

4. rsCLK wSRL - update the log (E=8)

5. wCDT.WRN - give a warning message (E=4) (ARC 1,2,3,4)

Task 15 : Critical Engine Temperature

Trigger Aperiodic - Minimum re-trigger time 1000 TU
Deadline 2000 TU **Processing Set**

1. wsFVL - close the fuel valves to shut off the engine (E=6)

2. wsCVL - open the coolant valves to maximum (E=6)

3. wEST - zero the speed in the engine state table (E=4)

4. wCDT.DIR - zero the speed in the display device window (E=4)

5. rsCLK wSRL - update the log (E=8)

6. wCDT.WRN - give a warning message (E=4) (ARC 1,2,3,4,5)

Task 16 : Read Engine 1 Consumption

Trigger Period 1000 TU
deadline 1000 TU **Processing Set**

1. rsCN1 wEET (E=8)

Task 17 : Read Engine 2 Consumption

Trigger Period 1000 TU
Deadline 1000 TU **Processing Set**

1. rsCN2 wEET (E=8)

Task 18 : Read Engine Temperature

Trigger Period 1000 TU
Deadline 1000 TU **Processing Set**

1. rsTMP wEET - read the temperature and save in the environment table (E=8)

2. rEET rEST wsCVL - read the temp and the desired temperature range and alter the state of the coolant valve as necessary (E=10)

Task 19 : Check Engine Speed

Trigger Period 1000 TU
Deadline 1000 TU **Processing Set**

1. rsES1 rEST wsFVL - change the fuel setting for the first engine (E=12)

2. rsES2 rEST wsFVL - change the fuel setting for the second engine (E=12)

Task 20 : Guidance System - check position

Trigger Period 2000 TU
Deadline 1000 TU **Processing Set**

1. rsBER wSLT - update the ship location table with the new location (E=8)
2. rOST - check the OST to make sure we are in autopilot (E=2) (ARC 1)
3. If in autopilot rSLT rSST - compare actual with required position (E=4)
4. If position reached rsCLK wSRL - update the ships log (E=8)
5. wSST - get the next position (E=4)
6. rSLT rSST wsRUD - alter the rudder to steer a true course (E=10)

Task 21 : Incoming message

Trigger Aperiodic - Minimum re-trigger time 1000 TU
Deadline 500 TU **Processing Set**

1. rsCOM rsCLK wCIL - update the communications input log with a new message header (E=10)
2. rsCOM wCIL - get the rest of the message (E=8)
3. rCIL (E=2)
4. If a mayday rsCLK wSRL - update the log (E=8)
5. wOST - put in manual mode (E=4)
6. rCIL rCTT rSLT wSST - generate a course to the distressed vessel (E=10) (ARC 5)
7. wsCOM - output a comms message to state intention to go to aid of vessel (E=6)
8. wCDT.WRN - output a warning message (E=4) (ARC 4,6,7)

Task 22 : Radar data ready

Trigger Period 1000 TU
Deadline 1000 TU **Processing Set**

1. rCTT wOTT - update the old track table (E=6)
2. rsRAD wCTT - generate the new set of points from the radar image (E=8)
3. rCTT rOTT rSLT rSST wCWT - check for any potential collisions (E=12)
4. rCTT rOTT rSLT rSST - check for unavoidable collisions (E=8)
5. If there are unavoidable collisions rCTT rOTT rSLT rSST wSRL - update the log (E=12)
6. rCTT rOTT rSLR rSST wCCT - update the collision critical table (E=12)
7. wEST - shut off the ships engines (E=4)
8. wsFLV - close off all fuel (E=6)
9. rCTT rOTT rSLT rSST wsCOM - output a mayday (E=14)
10. rCTT rOTT rSLT rSST wCDT.WRN - generate a warning message (E=12) (ARC 5,6,7,8,9)

Task 23 : Sonar data ready

Trigger Period 2000 TU **Deadline** 2000 TU **Processing Set**

1. rCTT wOTT - update the old state table (E=6)
2. rsSON wCTT - generate the new set of points from the radar image (E=8)
3. rCTT rOTT rSLT rSST wCWT - check for any potential collisions (E=12)
4. rCTT rOTT rSLT rSST - check for unavoidable collisions (E=8)

5. If there are unavoidable collisions rCTT rOTT rSLT rSST wSRL - update the log (E=12)

6. rCTT rOTT rSLR rSST wCCT - update the collision critical table (E=12)

7. wEST - shut off the ships engines (E=4)

8. wsFLV - close off all fuel (E=6)

9. rCTT rOTT rSLT rSST wsCOM - output a mayday (E=14)

10. rCTT rOTT rSLT rSST wCDT.WRN - generate a warning message (E=12)

The real-time tasks have now been defined in terms of actions on the real-time database. To summarise, the system consists of twenty three distinct real-time tasks.

B.3 Data Dependency Analysis

The independent tasks are linked by data dependencies at the transaction level. The transaction sets a re analysed and data dependency rings drawn, using the DDR CASE Tool (described in Appendix A), for each of the entities in the real-time system. For the sake of clarity, explicit representation of choice has been ommitted from the data dependency rings; the nested representation of transaction sets and selected subsets provides the information necessary to work out the scope of choice. These DDRs are shown in the first set of following figures.

Now that we have carried out the data dependency analysis, we can deduce the transaction precedence graph for each of the tasks. These graphs are shown in the following figures.

B.4 Transaction/Data Entity allocation

Given the transaction precedence graphs for each task and the data dependency rings, an allocation of transactions and data entities to a set of physical processors may be made. This section of the appendix describes such an allocation and how it was done. Given this allocation, a static analysis of the system may be carried out. A complete

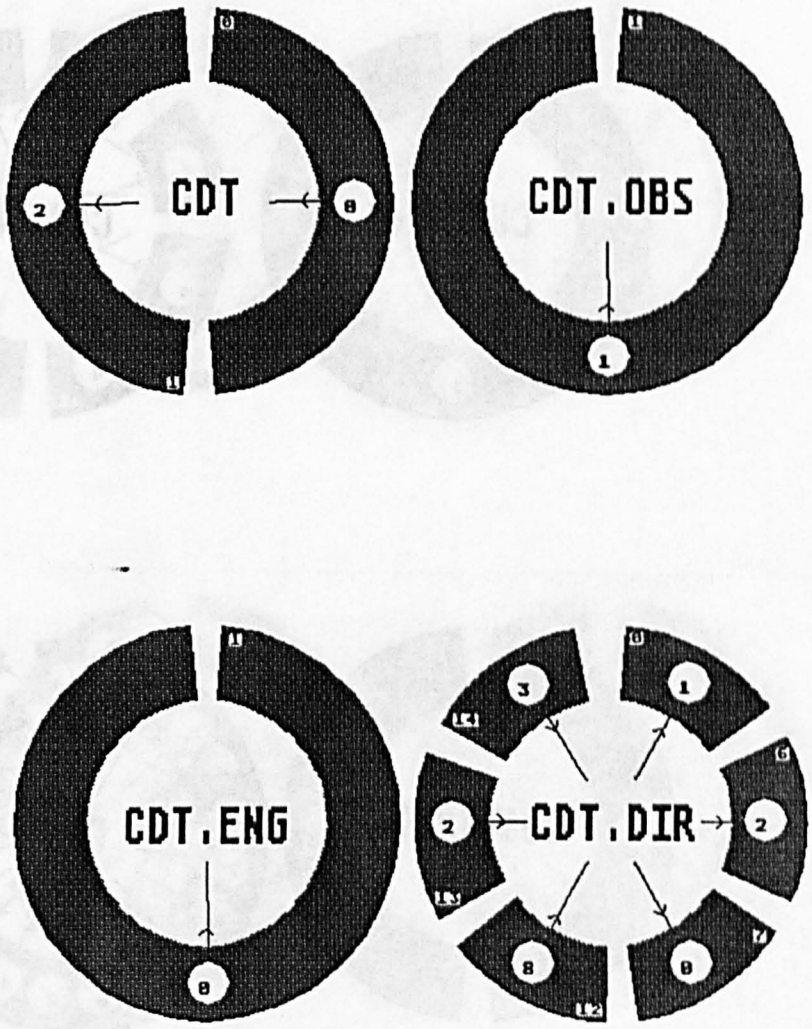

Figure B.4: Data Dependency Rings for the Ship Control System

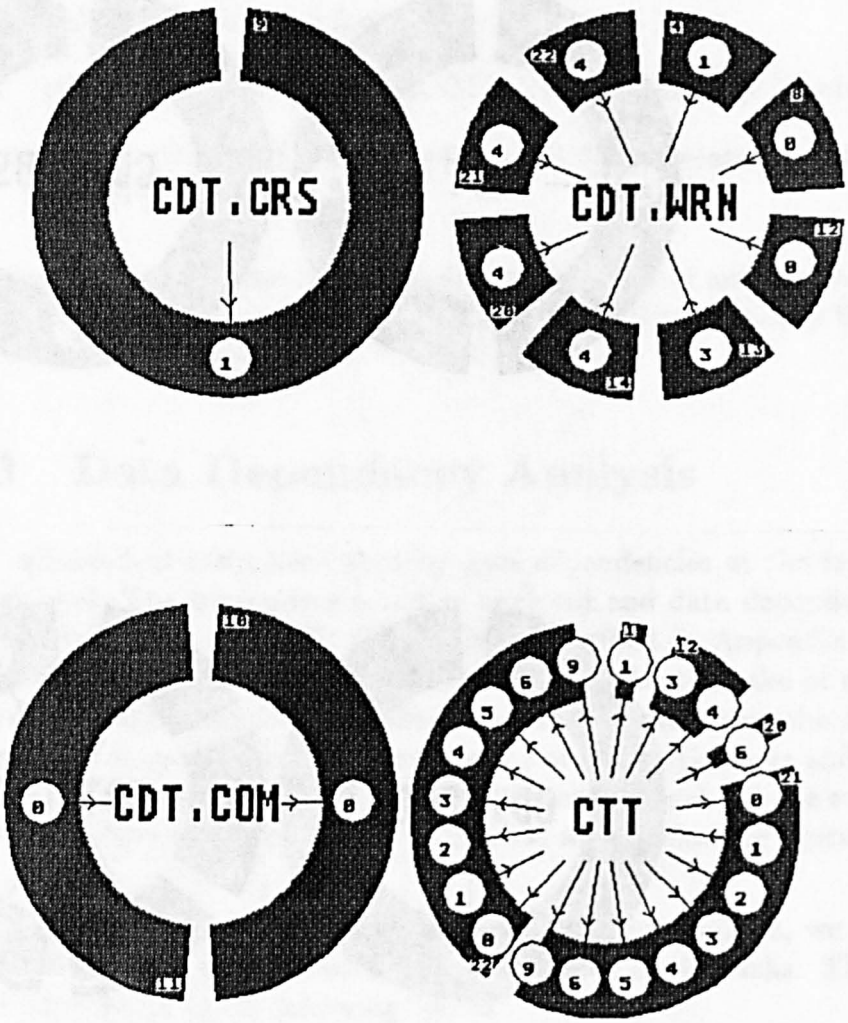

Figure B.5: Data Dependency Rings for the Ship Control System

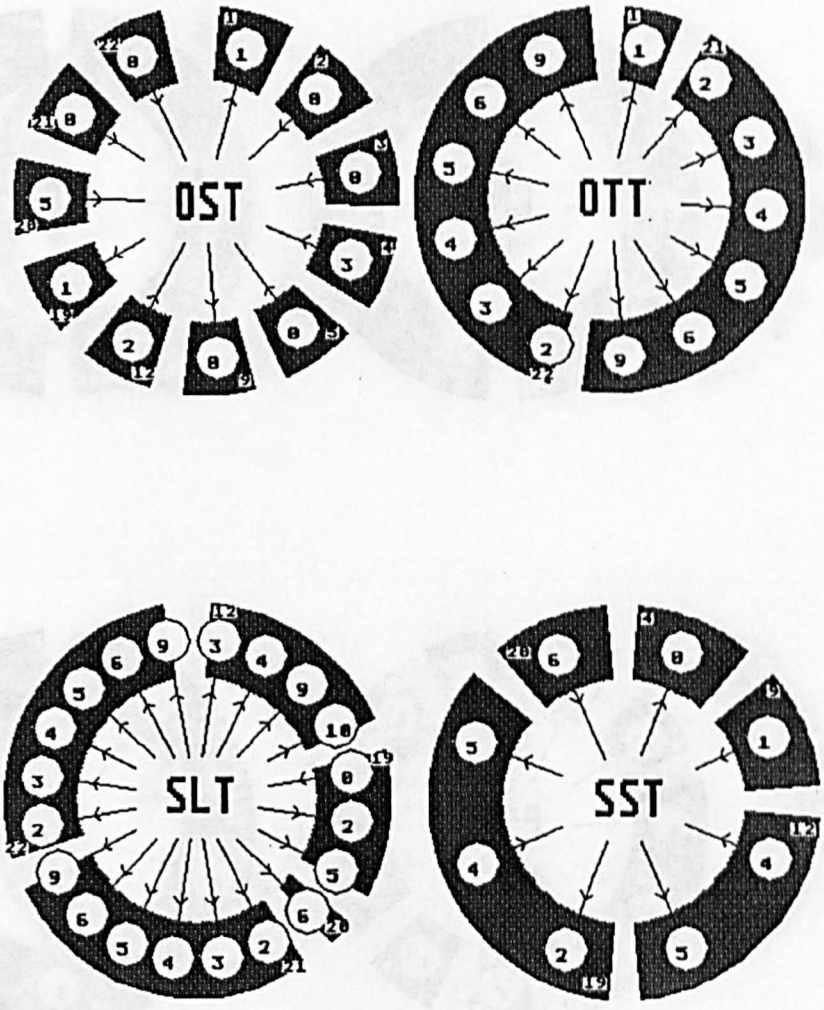

Figure B.6: Data Dependency Rings for the Ship Control System

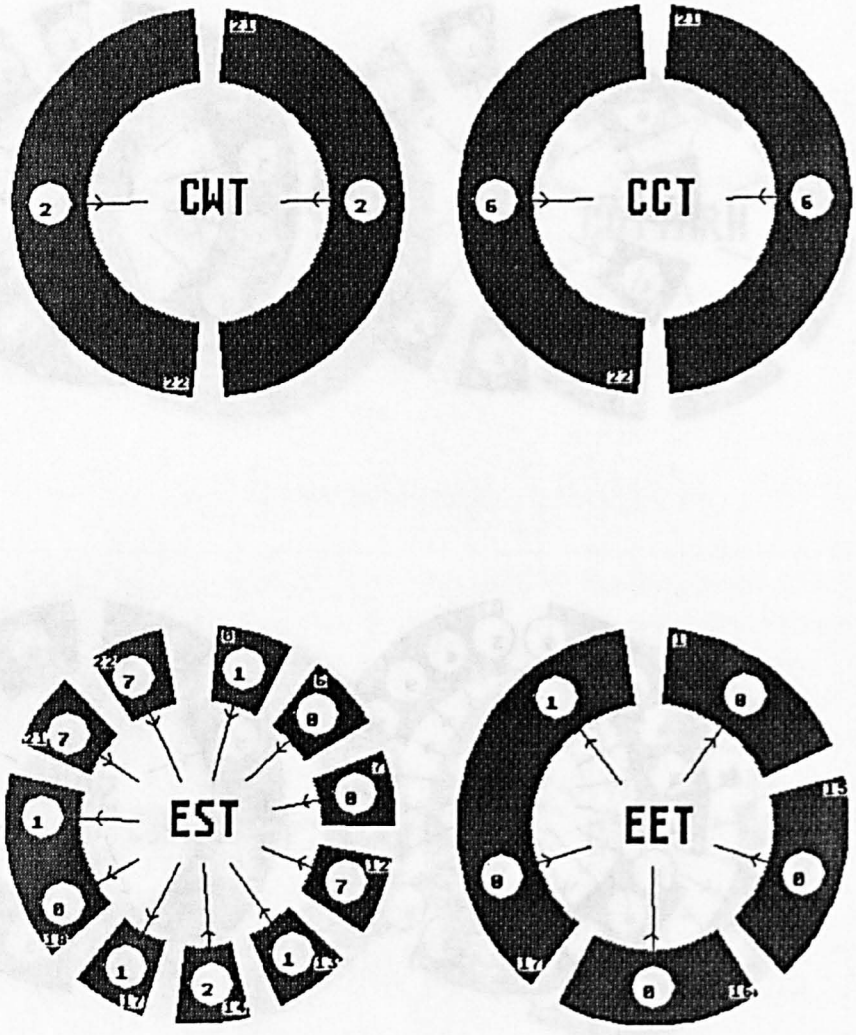

Figure B.7: Data Dependency Rings for the Ship Control System

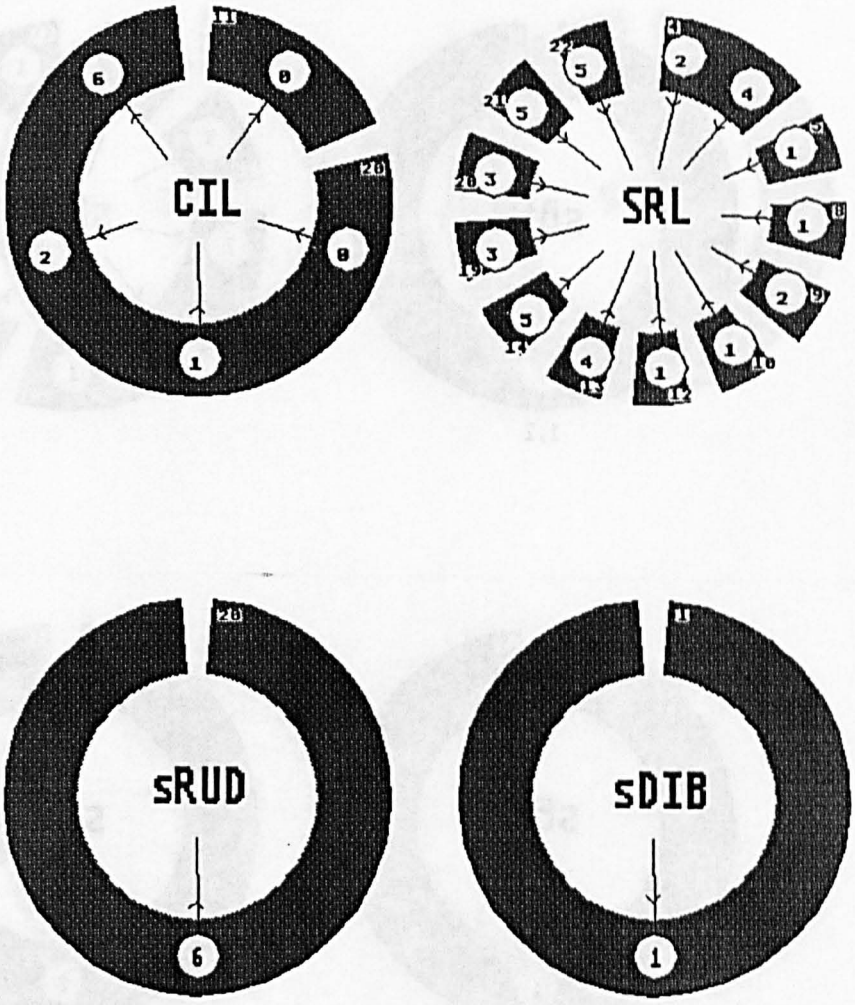

Figure B.8: Data Dependency Rings for the Ship Control System

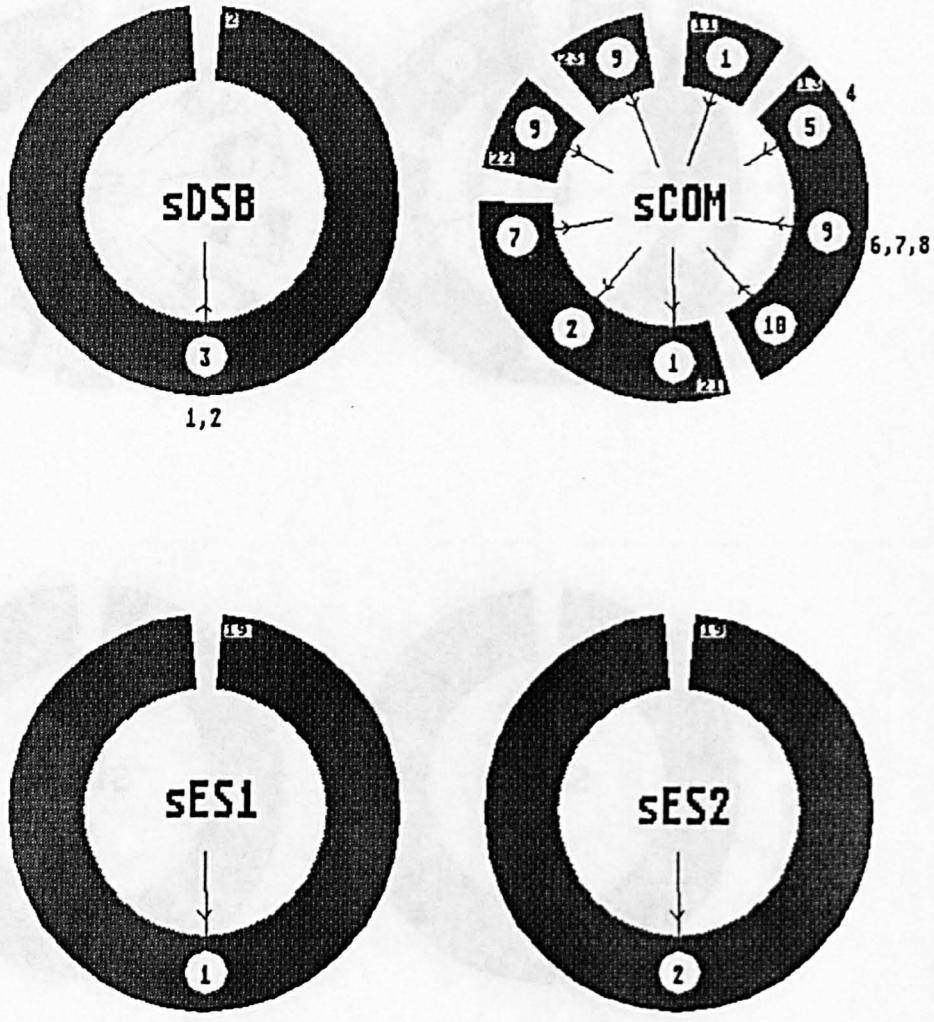

Figure B.9: Data Dependency Rings for the Ship Control System

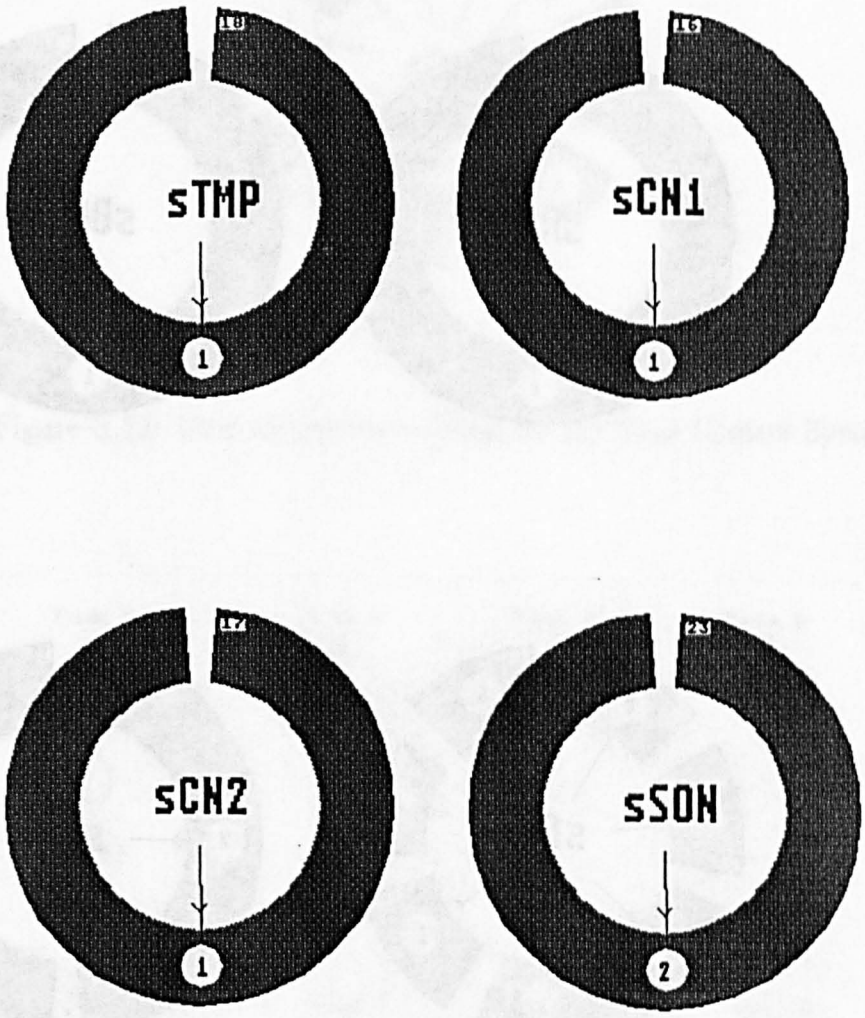

Figure B.10: Data Dependency Rings for the Ship Control System

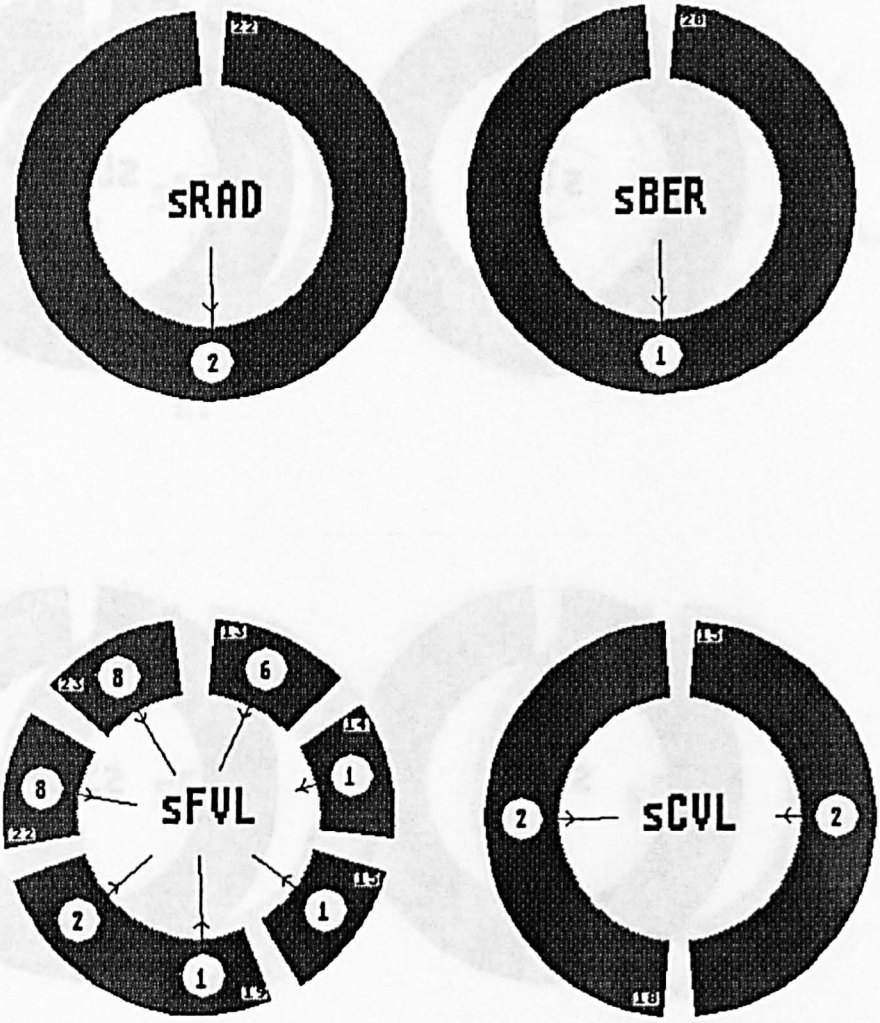

Figure B.11: Data Dependency Rings for the Ship Control System

Figure B.12: Data Dependency Rings for the Ship Control System

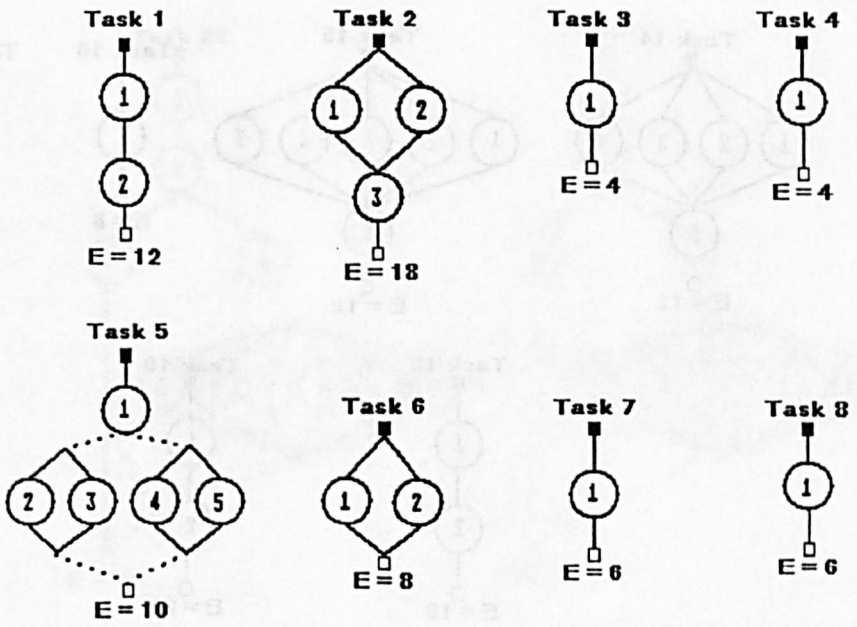

Figure B.13: Transaction Precedence Graphs : Tasks 1 to 8

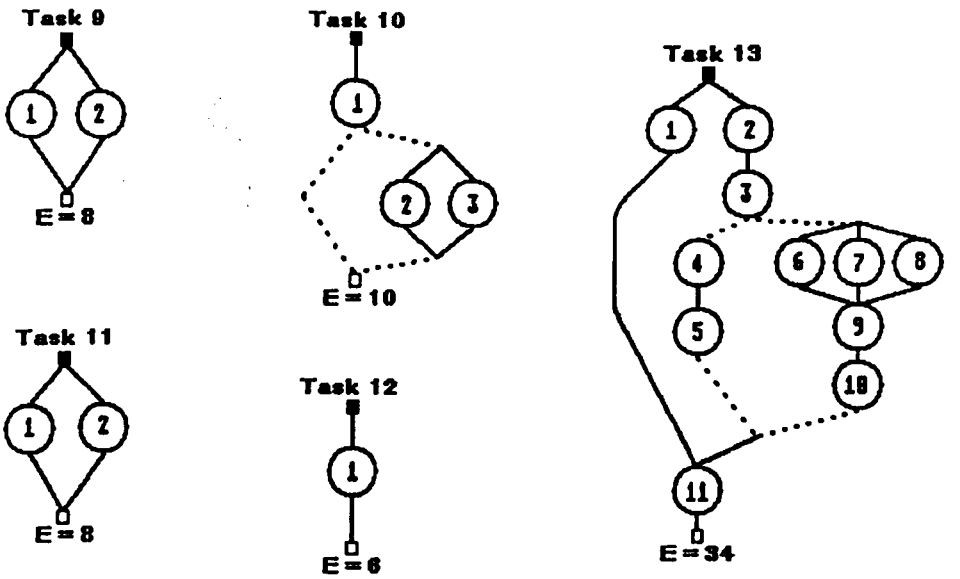

Figure B.14: Transaction Precedence Graphs : Tasks 9 to 13

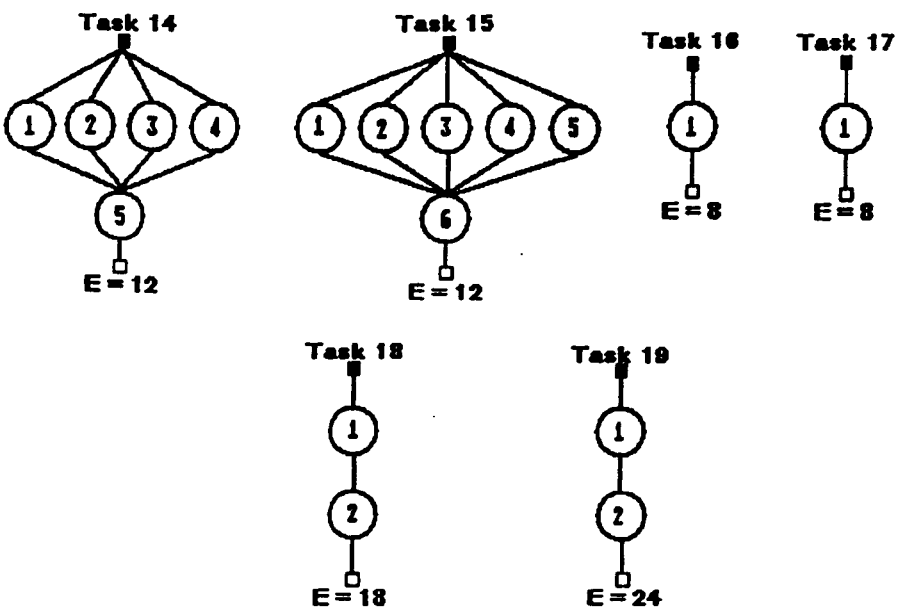

Figure B.15: Transaction Precedence Graphs : Tasks 14 to 19

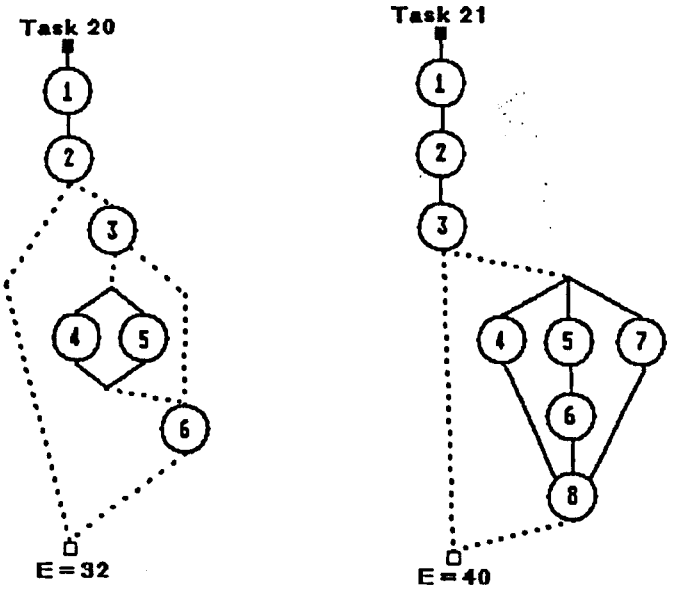

Figure B.16: Transaction Precedence Graphs : Tasks 20 to 21

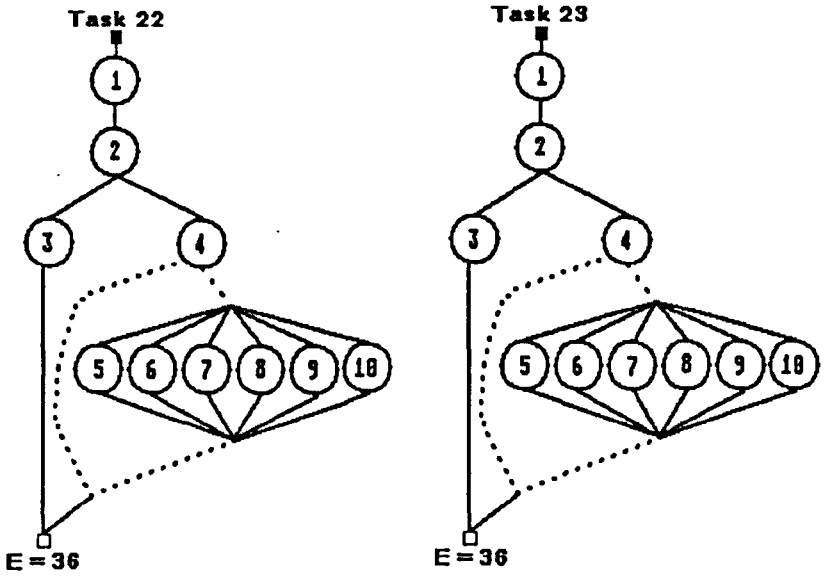

Figure B.17: Transaction Precedence Graphs : Tasks 22 to 23

static analysis is beyond this appendix. However, some static analysis examples are presented here to show the work that needs to be done to determine the schedulability of the implementation.

B.4.1 An allocation

Clusters and Logical Processors

Given the transaction precedence graphs for the real-time tasks we can construct a distribution of clusters and logical processors as described in Chapter 4. For each task, there is one cluster. Each cluster is made up of as many logical processors as are needed to make the best use of concurrency within the task. This concurrency is measured by the width of the transaction precedence graph for a task.

For example, consider the transaction precedence graph for task 9. Transactions 1 and 2 can execute concurrently so there will be two logical processors in the cluster for task 9; the first processor executes transaction 1 and the second executes transaction 2. Further consider the graph for task 10. In this task there are again two logical processors in the cluster. The first processor executes transactions 1 and then 2 and the second executes transaction 3 (concurrently with transaction 2). Transactions 1 and 2 can share a processor since they are sequential (sequence implied by the select construct in the graph).

This execise is now repeated for each of the processors. In the following cluster descriptions, each logical processor is called pX where X is an integer . For each logical processor, the transactions that are to be executed on it and the data entities that must be sited at it are specified in order that the transactions do not require any remote data access. The data entities are listed first; transactions are written as task number.transaction number.

Cluster 1
p1 sDIB CDT EST
1.1 1.2

Cluster 2
p1 EET CDY.ENG CDT sDSB
2.1 2.3 **p2** OST CTT OTT CDT.OBS
2.2

Cluster 3
p1 OST
3.1

Cluster 4
p1 OST
4.1

Cluster 5
p1 SST CDT.WRN OST
5.1 5.2 5.4
p2 sCLK SRL
5.3 5.5

Cluster 6
p1 OST
6.1
p2 sCLK SRL
6.2

Cluster 7
p1 CDT.DIR EST
7.1

Cluster 8
p1 CDT.DIR EST
8.1

Cluster 9
p1 CDT.WRN
9.1
p2 sCLK SRL
9.2

Cluster 10
p1 OST CDT.CRS SST
10.1 10.2
p2 sCLK SRL
10.3

Cluster 11
p1 CDT.COM sCOM
11.1
p2 sCLK SRL
11.2

Cluster 12
p1 CIL CDT.COM
12.1

Cluster 13
p1 sCLK SRL CDT.WRN

13.1 13.11
p2 OST CTT SLT SST sCOM sFVL
13.2 13.3 13.4 13.5 13.6
p3 EST SLT sCOM
13.7 13.9 13.10
p4 CDT.DIR
13.8

Cluster 14
p1 sFVL CDT.WRN
14.1 14.5
p2 EST
14.2
p3 CDT.DIR
14.3
p4 sCLK SRL
14.4

Cluster 15
p1 sFVL CDT.WRN
15.1 15.6
p2 sCVL
15.2
p3 EST
15.3
p4 CDT.DIR
15.4
p5 sCLK SRL
15.5

Cluster 16
p1 sCN1 EET
16.1

Cluster 17
p1 sCN2 EET
17.1

Cluster 18
p1 sTMP EET EST sCVL
18.1 18.2

Cluster 19
p1 sES1 EST sFVL sES2
19.1 19.2

Cluster 20

p1 sBER SLT OST SST sCLK SRL sRUD
20.1 20.2 20.3 20.4 20.6
p2 SST
20.5

Cluster 21
p1 sCOM sCLK CIL SRL CDT.WRN
21.1 21.2 21.3 21.4 21.8
p2 SST
20.5

Cluster 22
p1 CTT sRAD OTT SLT SST CWT
22.1 22.2 22.3
p2 CTT OTT SLT SST SRL
22.4 22.5
p3 CTT OTT SLR SST CCT
22.6
p4 EST
22.7
p5 sFLV
22.8
p6 CTT OTT SLT SST sCOM
22.9
p7 CTT OTT SLT SST CDT.WRN
22.10

Cluster 23
p1 CTT sSON OTT SLT SST CWT
23.1 23.2 23.3
p2 CTT OTT SLT SST SRL
23.4 23.5
p3 CTT OTT SLR SST CCT
23.6
p4 EST
23.7
p5 sFLV
23.8
p6 CTT OTT SLT SST sCOM
23.9
p7 CTT OTT SLT SST CDT.WRN
23.10

We have now reached the stage where for each task, we have defined a cluster of logical processors. For each processor within these clusters,

we have stated the transactions and data entities that must be resident for the maximum concurrency within the task to be achieved.

Reducing Logical Processors to Physical Processors

If we were to transfer the above distribution of transactions and logical processors directly to a physical network, we would need one physical processor for each logical processor. This amounts to 54 processors which is obviously very wasteful on such a simple application. Suppose the ship control system is required to use a maximum of 6 processors. We need to map the logical processors of the above clusters onto these 6 physical processors. This is carried out using the heuristics given in Chapter 4. These 'rules' (a) assign logical processors to physical processors such that as far as possible, logical processors from the same cluster as not assigned to the same physical processor (thus allowing concurrency within a task where possible) and (b) assign a logical processor to a physical processor that already has the required entities (or the greatest subset of them) already assigned.

This assignment of logical to physical processors relies on the judgement of the designer to some extent as well as the heuristic placement rules. As a result there are many different allocations to a physical processor set. The following is one such allocation. The notation is straight forward. For each physical processor, those logical processors allocated to it are listed. For example, physical processor 5 (P5) has the transactions and data entities from cluster 20 logical processor 2 (c20.p2) allocated to it.

Physical Processor P1 c1.p1, c2.p1, c7.p1, c8.p1, c13.p3, c14.p2, c15.p2, c15.p3, c16.p1, c17.p1, c18.p1, c19.p1, c22.p4, c23.p4
Physical Processor P2 c2.p2, c3.p1, c4.p1, c6.p1, c13.p2, c21.p2, c22.p1, c23.p1, c22.p5, c23.p5
Physical Processor P3 c5.p1, c10.p1, c14.p3, c15.p4, c20.p1, c22.p3, c23.p3, c22.p7, c23.p7
Physical Processor P4 c5.p2, c6.p2, c9.p2, c10.p2, c11.p2, c12.p1, c14.p4, c15.p5
Physical Processor P5 c20.p2, c22.p2, c23.p2
Physical Processor P6 c9.p1, c11.p1, c12.p1, c13.p4, c14.p1, c15.p1, c21.p1, c21,p3, c22.p6, c23.p6

This allocation is described in more detail listing in full the transactions and entities that are required at each of the physical processors in a distributed network implementation. These are shown in the following tables.

Processor 1		
Transactions	Entities	Devices
1.1 1.2	CDT	sDIB
2.1 2.3	EST	sDSB
7.1	EET	sCOM
8.1	SLT	sCVL
13.7 13.9 13.10	CTT	sCN1
14.2	OTT	sCN2
15.2 15.3	SST	sTMP
16.1	CDT.WRN	sES1
17.1		sFVL
18.1 18.2		sES2
19.1 19.2		
22.7 22.9 22.10		
23.7 22.9 23.10		

Processor 2		
Transactions	Entities	Devices
2.2	OST	sCOM
3.1	CTT	sFVL
4.1	OTT	sRAD
6.1	CDT.OBS	sSON
13.2 13.3 13.4 13.5 13.6	SLT	
21.5 21.6	SST	
22.1 22.2 22.3 22.8	CIL	
23.1 23.2 23.3 23.8	CWT	
	SRL	

Processor 3		
Transactions	Entities	Devices
5.1 5.2 5.4	SST	sBER
10.1 10.2	CDT.WRN	sCLK
14.3	OST	sRUD
15.4	CDT.CRS	
20.1 20.2 20.3 20.4 20.6	SST	
22.6 23.6	CDT.DIR	
	SLT	
	SRL	
	CTT	
	OTT	
	CCT	

Processor 4		
Transactions	Entities	Devices
5.3 5.5	CDT.WRN	sCLK
6.2	SRL	
9.2		
10.3		
11.2		
13.1 13.11		
14.4		
15.5		

Processor 5		
Transactions	Entities	Devices
20.5	SST	
22.4 22.5	CTT	
23.4 23.5	OTT	
	SLT	
	SRL	

Processor 6		
Transactions	Entities	Devices
9.1	CDT.WRN	sCOM
11.1	CDT.COM	sFVL
12.1	CIL	sCLK
13.8	CDT.DIR	
14.1 14.5	SRL	
15.1 15.6	OTT	
21.2 21.3 21.4 21.7 21.8		

Determining the Location of Scheduling Components

According to the execution environment in chapter 5, it is desirable to have one transaction scheduler for each task and one critical region scheduler for each data entity. The transaction scheduler implements a 'control token' based execution scheme to ensure the transactions are executed in the correct order; the critical region schedulers ensure that conflicting access to shared data entities is avoided. An important question now arises. Where are the transactions and critical region schedulers to be placed in the network of 6 processors. In order to reduce inter-processor communication as much as possible, the transaction scheduler for a task should be placed on that processor which has the most transactions for the task. Similarly, the critical region

scheduler for a data entity should be placed with that copy of the entity such that the controlling processor has the highest proportion of transactions that use the entity.

Using these two rules of thumb and using the data dependency rings to guide the placement of the critical region schedulers, we get the following allocation of scheduling components to processor in the implementation of the ship control system.

Placement of Transaction Schedulers to Processors					
Task	Processor	Task	Processor	Task	Processor
1	1	2	1	3	2
4	2	5	3	6	4
7	1	8	1	9	6
10	3	11	6	12	6
13	2	14	6	15	6
16	1	17	1	18	1
19	1	20	3	21	6
22	2	23	2		

Placement of Critical Region Schedulers to Processors					
Entity	Processor	Entity	Processor	Entity	Processor
CDT	1	CDT.OBS	1	CDT.ENG	1
CDT.DIR	1	CDT.CRS	1	CDT.WRN	1
CDT.COM	1	CTT	2	OST	2
OTT	2	SLT	3	SST	3
CWT	2	CCT	3	EST	1
EET	5	CIL	6	SRL	4
sRUD	3	sDIB	1	sDSB	1
sCOM	6	sES1	1	sES2	1
sTMP	1	sCN1	1	sCN2	1
sSON	2	sRAD	2	sBER	3
sFVL	2	sCVL	1	sCLK	4

B.4.2 Analysis

This section describes some of the analysis that is necessary to check the schedulability of the task set on the physical processors. The complete static analysis is long and involved and consequently beyond the scope of this appendix. This section demonstrates examples of the analysis that are required. Firstly, the 'raw' processing power of processor P5 is considered. Following this, the schedulability of Task 21 is considered.

Power of Processor P5

In considering an allocation of transactions to processors, it is important to check that the processor is powerful enough to execute all of the transactions before their deadlines, regardless of any problems caused by shared data between the transactions. For each shared resource, and this includes both processors and data entities, the 'raw processing power'[3] must be checked.

Consider the processing power of processor P5. This processor has parts of tasks 20, 22 and 23 allocated to it. We must first express these task parts as sub-tasks in their own right with their own execution times and deadlines.

The sub-task of task 20 has a trigger time 14 TU after the start of the whole task (that is after transactions 20.1, 20.2 and 20.3 have completed). The sub-task of task 20 must complete such that transaction 20.6 (with exectime of 10 TU) can execute before the deadline of the complete task. The deadline for the sub-task of task 20 is therefore 14+10 TU before the deadline of the complete task. The sub-task of task 20 therefore has an execution time of 4 TU and a deadline of 976 TU.

The sub-task of task 22 has a trigger time of 14 TU after the start of the whole task (that is afetr transactions 22.1 and 22.2 have completed). Since no transactions from task 22 follow those allocated to processor P5, the deadline for the sub-task of task 22 is therefore 14 TU before the deadline of the complete task. The sub-task of task 22 therefore has an execution time of 20 TU and a deadline of 986 TU.

The sub-task of task 23 has a trigger time of 14 TU after the start of the whole task (that is afetr transactions 23.1 and 23.2 have completed). Since no transactions from task 23 follow those allocated to processor P5, the deadline for the sub-task of task 23 is therefore 14 TU before the deadline of the complete task. The sub-task of task 23 therefore has an execution time of 20 TU and a deadline of 1986 TU.

Given these characteristics, and assuming that the minimum re-trigger times for the tasks are at the deadlines calculated, we can calculate a common re-trigger time of 477799284 TU. This represents 489549 triggering of task 20 (execution time on this processor of 1958196

[3]It doesn't really make much sense to refer to the raw processing power of a data entity; the term is meant to convey the idea of the ability of all transactions to use the resource before their deadlines while ignoring the restriction that transactions must be executed as an atomic unit. If the resource fails this test then there is no point in continuing with the static analysis

TU); 484584 triggerings of task 22 (execution time on this processor of 9691680 TU) and 240584 triggerings of task 23 (execution time on this processor of 4811680). The total execution time of 16461556 is less than the common re-trigger time calculated and hence these tasks pass the first test: there is enough raw processing power to complete the tasks before their deadlines.

This type of reasoning should be repeated for each of the other processors and also for any other shared resource e.g. shared data entities and hardware devices. Even from this simple example, it is seen that the numbers generated by this stage of the analysis may be prohibitively large.

Schedulability of Task 21

On successful completion of the first stage of the analysis, a more detailed examination of each of the tasks can be carried out. This examination tests to see that the tasks are schedulable using a particular scheduling policy. The first set of tests should check that the scheduling policy works for the tasks on each of the processors given that the tasks should execute as indivisible units. The second set of tests checks that each task can meet its deadlines given the scheduling policy and recognising that each critical region within the task must be executed as an indivisible unit. This last series of tests check that the critical regions on each entity can be serialised such that each meet their deadline. For the EDF scheduling policy, each of these tests are broken into two parts; the first considers the case when the tasks are non-preemptable, the second considers the case when the tasks are pre-emptable (once only for the sake of argument).

The first part of the tests is demonstrated for Task 21. Task 21 is located on processors P2 and P6. For each of these processors, we must describe the transactions that are placed on them in terms of ammended deadlines and re-trigger times. As in the tests to validate the raw processing power, the deadline of a sub-task on a processor is brought closer based on the amount of work the task has completed when it started the transactions on this processor and how much work after these transactions there is left to complete. For the tasks on processor P2 we have the the following ammened task characteristics:

Ammended Task Characteristics for P2			
Task	Execution Time	Deadline	M.R.T.
2	10	12	20
3	4	20	40
4	4	20	40
6	4	8	40
13	24	1996	2000
21	9	476	1000
22	26	1000	1000
23	26	2000	2000

With non-preemptive EDF scheduling, the worst case scenario for task 21 occurs when it is triggered just as a task with the longest execution time starts executing on this processor. This might be either task 22 or 23 since they both cause task 21 to wait for 26 TUs. The worst case for the task then continues with all those tasks with sooner deadlines continually re-triggering. The soonest time that task 21 can begin executing can be calculated and thus it can be determined in this worst case, whether the deadline is met or not. For this example, the worst case consists of the 26 TU delay already mentioned as well as re-triggerings of tasks 2,3,4 and 6. Even a cursory examination of these tasks will show that task 21 cannot meet its deadline. Indeed, simultaneous triggerings of tasks 2 and 6 cannot even meet both deadlines. The analysis shows that the tasks on this processor do not meet their deadlines and corrective action needs to be taken.

We now demonstrate the similar test to show that a set of critical regions on a data entity can be scheduled correctly. Consider the Engine Environment Table EET. Constructing the sub-tasks that must be scheduled through this data entity we get the following :

Sub-tasks to be scheduled through the EET			
Task	Exectime	Deadline	M.R.T
2	6	12	20
16	8	1000	1000
17	7	1000	1000
18	18	1000	1000

Let us consider the non-preemptive case. The worst case delay that task 2 suffers from contention on this data entity is when it has to wait for task 18 to complete its use of the entity. This delay is 18 TU. Since the slack of task 2 cannot accommodate this delay, there is a problem that must be resolved. The worst case delay that task 16 suffers occurs when it has to first of all wait for task 18 to complete (18 TUs) and

then wait for one occurance of task 2 (with its sooner deadline) to finish (6 TUs). This delay is 24 TUs; the slack of task 16 can accommodate this delay.

In the preemptive case, task 2 does not have to suffer any delay; the task can always begin executing, until it gets to the stage where a given task has been preempted by task 2 too many times and the system moves over to a non-preemptive scheduler to give the preempted task processor time. As an example of one of the other tasks consider task 18. The worst case for this is if it is interrupted just as it was about to complete, by a triggering of task 2. The delay is thus 18 TUs (for the aborted execution of the task) plus 6 TUs (triggering of task 2) i.e. 24 TUs. This can easily be accommodated within the slack of the task.

B.5 Conclusions

The example described in this appendix shows the complete development of a relatively large real-time database system from its high level description through to the allocation of transactions and data entities in a distributed replicated database environment. Hints, as to the static analysis that is required to prove the schedulability of the real-time tasks are given.

The example clearly demonstrates the applicability of the design methodology. Each stage of the method had a set of deliverables that were used in the next stage to further the development. The first main stage of the methodology, setting up the context diagrams and describing the database entities was fairly straigtforward. The CASE tool described in the previous appendix was used to generate the context diagram of figure B.1. The next stage; describing the real-time tasks in terms of actions on this database proved to be the stage of the method that was most open to problems. The final design is very much dependent on the success of this stage; other stages from this being mechnanical transformations from this stage. For the simple ship control example, the transaction decomposition was relatively painless however. The CASE tool was then used to generate the data dependency rings and transaction precedence graphs shown in the previous figures.

The allocation scheme illustrated in the example was the one described in chapters 4 and 5. Although a simple process, the scheme suffers from the problem of uneven static load balancing. This is illustrated in the large number of transactions allocated to processor 1 and less allocated to processor 5 for example.

The static schedulability analysis illustrated at the end of the design exercise highlights a further flaw in the method. This is that the static analysis is a very laborious task. Indeed, in the example, a very small part of the necessary analysis was carried out. The CASE tool offers some support for the analysis, but it is, by no means, complete.

Appendix C

An analysis Example

In this appendix we present a simple, but complete, static analysis of an example task set to check whether or not the task set is sound. The analysis has the following steps.

1. Check serialisation through processors
2. Check serialisation through data entities
3. Check shared data - no preemption
4. Check shared data - preemption and backoff

Step 1 checks that there is enough CPU time to fully execute the tasks assuming that the task set is fully preemptable and that the processor is the only shared resource. If the task set is unsound in this, the best case, there is no point in further analysis. Step 2 checks that there is enough CPU time to meet the use of each data entity. This step treats the data entity as if it were a processor and each task using this is preemptable. If the task set is unsound at this stage, then it is impossible for the task set to make serial, non-preemtable, use of the data entity. Steps 3 and 4 check to see that a valid schedule is possible for tasks using each data entity. This schedule provides each task with uninterrupted use of the data entity.

For consiseness, the temporal properties of a task may be described by the tuple

Task no.(execution time, deadline, MRT)

Consider the following task set.

- 1(2,10,10) has critical regions on A
- 2(3,10,10) has critical regions on AB
- 3(5,20,20) has critical regions on B
- 4(3,15,15) has critical regions on C
- 5(2,10,10) has critical regions on C

A task/data entity allocation scheme has been proposed. Tasks 1,2 and 3 together with data entities A and B are placed on processor 1. Tasks 4 and 5 and data entity C are placed on processor 2. The static analysis for the suitability of EDF scheduling on this task set is carried out as follows:

Check Serialisation Though Processors

Processor 1

```
Earlist Common Retrigger Time = 20 TU
No of triggers of Task 1 = 20/10 = 2 = 4 TU
"  "     "     "   "   2 = 20/10 = 2 = 6 TU
"  "     "     "   "   3 = 20/20 = 1 = 5 TU
                                    ------
                                     15

15 TU <= 20 TU
Processor 1 okay
```

Processor 2

```
Earlist Common Retrigger Time = 30 TU
No of triggers of Task 4 = 30/15 = 2 = 6 TU
"  "     "     "   "   5 = 30/10 = 3 = 6 TU
                                    ------
                                     12

12 TU <= 30 TU
Processor 2 okay
```

Check Serialisation Through Shared Data

Data Entity A

```
Earlist Common Retrigger Time = 10 TU
No of triggers of Task 1 = 10/10 = 1 = 2 TU
"  "     "     "   "   2 = 10/10 = 1 = 2 TU
                                      ------
                                        4

4 TU <= 10 TU
Data Entity A okay
```

Data Entity B

```
Earlist Common Retrigger Time = 20 TU
No of triggers of Task 2 = 20/10 = 2 = 4 TU
"  "     "     "   "   3 = 20/10 = 1 = 5 TU
                                      ------
                                        9

9 TU <= 20 TU
Data Entity B okay
```

Data Entity C

```
Earlist Common Retrigger Time = 30 TU
No of triggers of Task 4 = 30/15 = 2 = 6 TU
"  "     "     "   "   5 = 30/10 = 1 = 6 TU
                                      ------
                                       12

12 TU <= 30 TU
Data Entity C okay
```

Check For Shared Data With No Preemption

Task 1

```
Worst case delay is when T1 is triggered just after T2
starts executing.
delay = 3 TU (exec time of T2) <= 8 TU (slack of T1)
Task 1 okay in the worst case.
```

Task 2

```
Worst case delay is when T2 is triggered just after T3
starts executing.
delay = 5 TU (exec time of T3) <= 7 TU (slack of T2)
Task 2 okay in the worst case.
```

Task 3

```
Worst case delay is when T3 is triggered just after T2
starts executing.
delay = 3 TU (exec time of T2) <= 15 TU (slack of T3)
Task 3 okay in the worst case.
```

Task 4

```
Worst case delay is when T4 is triggered just after T5
starts executing.
delay = 2 TU (exec time of T5) <= 12 TU (slack of T4)
Task 4 okay in the worst case.
```

Task 5

```
Worst case delay is when T5 is triggered just after T4
starts executing.
delay = 3 TU (exec time of T5) <= 8 TU (slack of T5)
Task 5 okay in the worst case.
```

Check for Shared Data With Preemption and Backoff

Assume that a task may be preempted once only.

Task 1

```
If task 1 is started then it will execute through to the end since
no other task triggered after task 1 starts can have a deadline
before that of task 1.
```

Task 2

```
If task 2 is started then it executes through to the end since
no other task triggered after task 2 starts can have a deadline
before that of task 2.
```

Task 3

```
Worst case is when T3 is preempted by T2 (which requires access
to B) just before it is due to complete and then also by T1
(which requires access to the shared processor).
```

```
Worst Case = 5 (aborted exec of T3) + 3 (exec of T2) +
             2 (exec of T1)
           = 10 TU
Deadline of T3 - Worst Case Delay = 20-10 = 10 TU >= exec time
of T3

Task T3 okay
```

Task 4

```
Worst case is when T4 is preempted by T5 (which requires access
to C) just before it is due to complete.

Worst Case = 3 (aborted exec of T4) + 2 (exec of T5)
           = 5 TU
Deadline of T4 - Worst Case Delay = 15-5  = 10 TU >= exec time
of T4

Task T4 okay
```

Task 5

```
If task 5 is started then it executes through to the end since
no other task triggered after task 5 starts can have a deadline
before that of task 5.
```

The analysis shows that the simple task set and allocation is sound using the EDF scheduling policy. Both non-preemption and a single preemption and backoff will work such that all deadlines of the five tasks are met in the worst case.

Appendix D

Example Execution Traces For The Scheduler Hierarchy

To demonstrate how the hierarchy of scheduling mechanisms of chapter 5 works in this appendix we present an example. Suppose the real-time system has two tasks, T1 and T2. The transaction precedence graphs for the tasks are shown in figure D.1.

In the implementation, it is decided to allocate two processors P1 and P2 to task T1 and processor P3 to task T2. Each of these processors has a copy of the shared data entity A. Processors P1 and P2 are dedicated to executing task T1 and processor P3 is dedicated to T2. Processor P1 executes transactions T1.1 and T1.2; processor P2 executes transactions T1.3 and T1.4; processor P3 executes transactions T2.1 and T2.2. Each task cluster has a separate transaction scheduler; there is also a task and critical region scheduler in the system. Static analysis of the temporal requirements of the tasks shows that the EDF scheduling policy is sound. The allocation scheme is shown in figure D.2. The locations of the scheduler mechanisms are not considered in this example.

The following scenario depicts the course of events if task T1 is triggered, and during its execution lifetime, task T2 remains idle. The scenario describes the information flow through the scheduling mechanism. This information flow is characterised as being either

- Event Flow E shows the triggering of the tasks.
- Control Flow C shows the scheduler signals that start and stop tasks.
- Update propagation U shows the updates being applied to non-

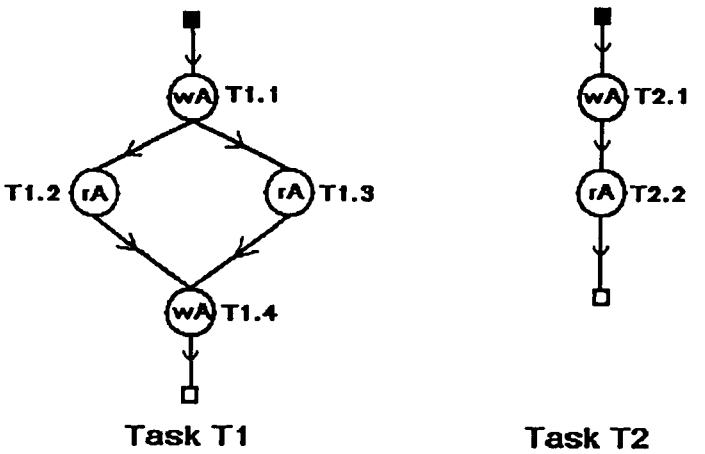

Figure D.1: Transaction Precedence Graphs for tasks T1 and T2

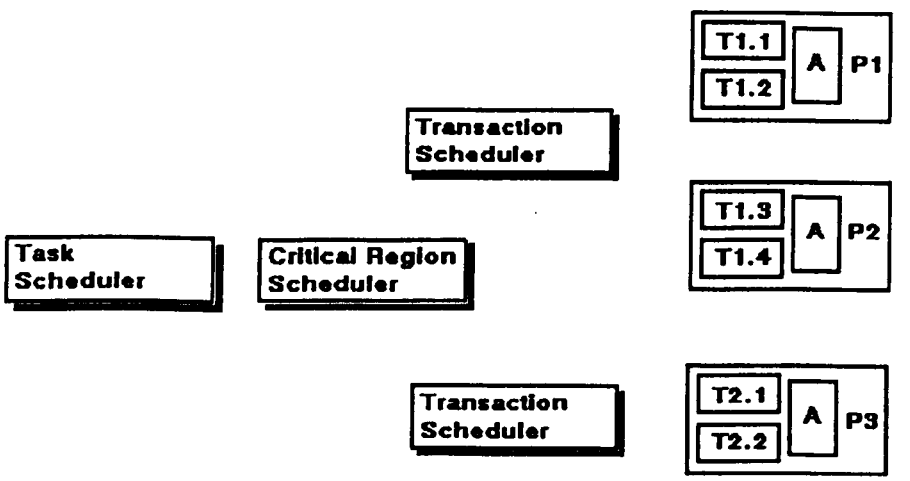

Figure D.2: The Allocaton of Tasks, Schedulers and Data Entities to Processors

local copies of shared data entities.

- Data flow **D** Shows data flowing from one point in the system to another.

In describing the scenario, the type of the information flow at each stage is shown. Execution proceeds as follows:

1. (E). Task T1 is triggered by some event in the environment. The task scheduler intercepts this trigger.
2. (C). The task scheduler determines the correct token 'colour' for this invocation of task T1 and sends this to the appropriate transaction scheduler.
3. (C). The transaction scheduler starts to execute task T1. A list of the four T1 transactions is created, 'coloured' with the control token to identify this execution list from any other current invocation of the task. Each member of the list has the predecessors listed. The transaction scheduler scans the list and recognises that T1.1 waits for no other transaction. The transaction scheduler sends a request message to the critical region scheduler to gain write access to entity A.
4. (C). The critical region scheduler checks the lock table for A and sends a control signal back to the transaction scheduler to show that T1 may enter the critical region on A.
5. (C). The transaction scheduler sends a control message to T1.1 to start executing.
6. (D). Transaction T1.1 updates its own local copy of entity A.
7. (CD). Transaction T1.1 sends a completed control signal to the transaction scheduler. This signal includes the updates T1.1 has made to entity A.
8. (U). The transaction scheduler applies the updates to the other copies of entity A used within this cluster (task) and then updates the transaction list to show that T1.1 has completed.
9. (C). The transaction scheduler sends control signals to transactions T1.2 and T1.3 to start them executing.
10. (D). Data flows from entity A into transaction T1.2.
11. (D). Data flows from entity A into transaction T1.3.

12. (C). Transaction T1.2 sends a control signal to the transaction scheduler to show that it has finished.

13. (C). Transaction T1.3 sends a control signal to the transaction scheduler to show that it has finished.

14. (C). The transaction scheduler updates the transaction list for T1 and then sends a control signal to T1.4 to start it executing.

15. (D). Transaction T1.4 updates its own local copy of entity A.

16. (CD). Transaction T1.4 sends a completed control signal to the transaction scheduler. This signal includes the updates T1.4 has made to entity A.

17. (U). The transaction scheduler applies the updates to the other copies of entity A used within this cluster (task) and then updates the transaction list to show that T1.4 has completed.

18. (CD). The transaction scheduler checks the transaction list and finds that task T1 has completed. It sends the updates made to the shared entity A to the critical scheduler.

19. (U). The critical region scheduler propagates the updates to the other copies of entity A used by other tasks. The critical region scheduler releases the lock on entity A.

20. (C). The transaction scheduler sends a control signal to the task scheduler to show that task T1 has finished executing.

This execution of task T1 is illustrated in figure D.3.

We now consider the scenario where task T1 is interrupted by task T2. Suppose task T2 has a sooner deadline than T1; T2 therefore takes priority. The following information flow takes place in the system.

1. (E). Task T1 is triggered by some event in the environment. The task scheduler intercepts this trigger.

2. (C). The task scheduler determines the correct token 'colour' for this invocation of task T1 and sends this to the transaction scheduler.

3. (C). The transaction scheduler starts to execute task T1. A list of the four T1 transactions is created 'coloured' with the control token to identify this execution list from any other current invocation of the task. Each member of the list has the predecessors

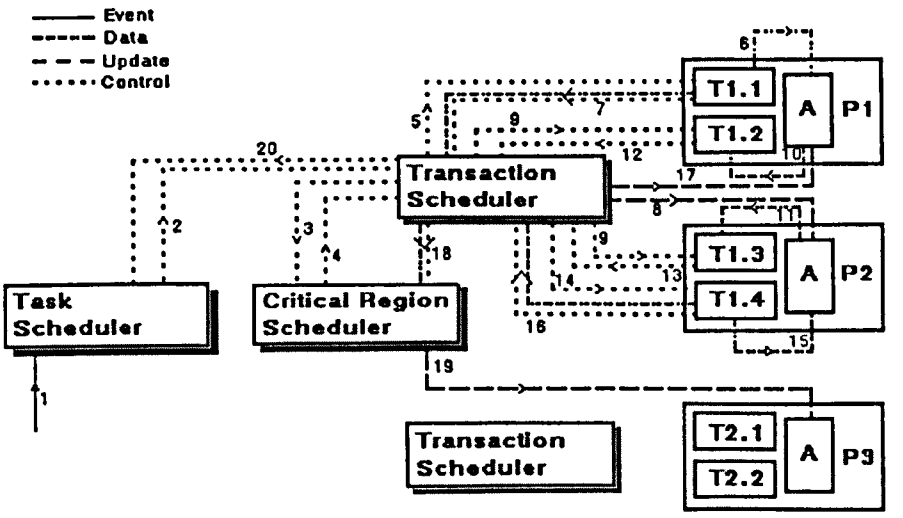

Figure D.3: Execution of task T1

listed. The transaction scheduler scans the list and recognises that T1.1 waits for no other transaction. The transaction scheduler sends a request message to the critical region scheduler to gain write access to entity A.

4. (C). The critical region scheduler checks the lock table for A and sends a control signal back to the transaction scheduler to show that T1 may enter the critical region on A.

5. (C). The transaction scheduler sends a control message to T1.1 to start executing.

6. (D). Transaction T1.1 updates its own local copy of entity A.

7. (CD). Transaction T1.1 sends a completed control signal to the transaction scheduler. This signal includes the updates T1.1 has made to entity A.

8. (U). The transaction scheduler applies the updates to the other copies of entity A used within this cluster (task) and then updates the transaction list to show that T1.1 has completed.

9. (C). The transaction scheduler sends control signals to tran sactions T1.2 and T1.3 to start them executing.

10. (D). Data flows from entity A into transaction T1.2.

11. (D). Data flows from entity A into transaction T1.3.

12. (E). Task T2 triggers. The task scheduler intercepts this trigger.

13. (C). The task scheduler determines the correct token 'colour' for this invocation of task T2 and sends this to the transaction scheduler for task 2.

14. (C). The transaction scheduler starts to execute task T2. A list of the two T2 transactions is created 'coloured' with the control token to identify this execution list from any other current invocation of the task. Each member of the list has the predecessors listed. The transaction scheduler scans the list and recognises that T2.1 waits for no other transaction. The transaction scheduler sends a request message to the critical region scheduler to gain write access to entity A.

15. (C). The critical region scheduler checks the lock table for A and sees that task T1 is already using it. The EDF scheduling policy is then used and it is decided that task T2 should go ahead. The critical region scheduler sends a control message to the transaction scheduler of T1 to stop execution of T1. This transaction scheduler will then clear the transaction list for T1.

16. (C). The critical region scheduler sends a control message to the transaction scheduler of T2 to start it executing.

17. (C). The transaction scheduler starts T2.1 executing.

18. (D). Transaction T2.1 updates its own local copy of entity A.

19. (CD. Transaction T2.1 sends a completed control signal to the transaction scheduler. This signal includes the updates T2.1 has made to entity A.

20. (C). The transaction scheduler updates the transaction list for task T2 and then sends a control signal to T2.2 to start it executing.

21. (D). Data flows from entity A into transaction T2.2

22. (C). Transaction T2.2 sends a completed control signal to the transaction scheduler. The transaction scheduler checks the transaction list and recognises that this is the end of task T2.

23. (CD). The transaction scheduler for T2 sends a control signal to the critical region scheduler to show that task T2 has finished. The updates that T2 made to A are also passed onto the critical region scheduler.

24. (U). The critical region scheduler removes the lock from the lock table and sends the updates to other copies of entity A used by other tasks.

25. (C). The critical region scheduler sends a control message to the transaction scheduler of task 1 to restart task 1. The execution of T1 then procedes uninterrupted as in the previous scenario.

The example execution traces presented show how for a simple task set, the hierarchy of scheduling mechanisms ensure the serial execution of conflicting transactions on shared data. The examples show that the scheduling hierarchy makes it simple to add further tasks to the system by providing an appropriate transaction scheduler.