THESIS REPORT

Master's Degree

Modeling and Analysis of Real-Time Database Systems in the Framework of Discrete Event Systems

by A. Ghosh
Advisor: S.I. Marcus

M.S. 95 -6

Sponsored by
the National Science Foundation
Engineering Research Center Program,
the University of Maryland,
Harvard University,
and Industry

Abstract

Title of Thesis:

Modeling and Analysis of
Real-Time Database Systems
in the Framework of
Discrete Event Systems

Name of degree candidate: Anunoy Ghosh

Degree and year: Master of Science, 1994

Thesis directed by: Professor Steven I. Marcus
Department of Electrical Engineering

Associate Professor Mark Shayman
Department of Electrical Engineering

Real-time systems are an active area of research currently, motivated by the potential of widespread applicability in areas like stock trading, network management, air traffic control, robotics and factory automation. Since these systems deal with large quantities of information, real-time systems are being coupled with database systems to aid in the efficient storage, processing and retrieval of data. Such database systems are called Real-Time Database Systems (RTDBS).

The problem of concurrency control and scheduling of transactions in real time database systems is studied in the framework of discrete event dynamical systems (DEDS) modeled by deterministic finite automata (DFAs). Concurrency control and scheduling are separated into two different modules (a logical DEDS model for the CC module and a heuristic implementation of a scheduler) to allow modular analysis of various combinations of concurrency control and scheduling algorithms. The model is developed analytically using the theory of discrete event dynamical systems. Subsequently the design of a simulation software is reported that uses this model to simulate transaction execution for a (concurrency controller, scheduler) pair. Finally, we show that our approach can also be viewed as a special case of a supervisory control theory (SCT) synthesis technique. The goal of this thesis is to demonstrate the applicability of DEDS theory as a powerful tool in modeling and analyzing transaction models in real time database systems and to show potential applications of modern SCT techniques in this area.

Modeling and Analysis of Real-Time Database Systems in the Framework of Discrete Event Systems

by

Anunoy Ghosh

Thesis submitted to the Faculty of the Graduate School
of The University of Maryland in partial fulfillment
of the requirements for the degree of
Master of Science
1994

Advisory Committee:

Professor Steven I. Marcus, Chairman/Advisor
Associate Professor Mark Shayman
Assistant Professor Anindya Datta

Acknowledgements

I have been highly privileged to work under the supervision of Professor Steven Marcus and Professor Mark Shayman. I would like to thank them for introducing me to the field of Discrete Event Systems and Supervisory Control.

My decision to examine real-time database systems in the framework of discrete event systems and supervisory control resulted from my discussions with Professor Anindya Datta during his visit to the Institute for Systems Research. It is a pleasure to acknowledge his patient guidance while I was developing the simulation software RTSIM.

I am grateful for the help I have received from the students and staff in the Systems Integration Laboratory. I enjoyed our many discussions and the exchange of ideas and knowledge about specific technical issues which I am sure will prove invaluable in the future.

Finally, I owe a priceless debt to my parents, Maya and Utsa, whose encouragement has always made me strive harder towards achieving my goals.

I gratefully acknowledge financial support from NSF (grant number NSFD CDR 8803012) and the Air Force (grant number F 49620-92-J-0045), and research assistantships from Professor Marcus.

Table of Contents

List of Figures

Modeling and Analysis of
Real-Time Database Systems
in the Framework of
Discrete Event Systems

Anunoy Ghosh

July 17, 1995

This comment page is not part of the dissertation.

Typeset by LaTeX using the `dissertation` style by Pablo A. Straub, University of Maryland.

Chapter 1

Introduction

1.1 Real-Time Database Systems

Real-time systems are an active area of research currently [68], motivated by the potential of widespread applicability in areas like stock trading, network management, air traffic control, robotics and factory automation. Since these systems deal with large quantities of information, real time systems are being coupled with database systems to aid in the efficient storage, processing and retrieval of data. Such database systems are called Real-Time Database Systems (RTDBS) [60, 69].

Conventional databases deal with persistent data. Transactions access this data while maintaining its consistency. Serializability [15] is the usual correctness criterion associated with such transactions. In contrast, a RTDBS deals with temporal data, i.e., data that becomes outdated after a certain time. The temporal nature of the data and the response time requirements imposed by the environment cause timing constraints such as periods or deadlines to be associ-

ated with the transactions. The resulting important difference is that the goal of a RTDBS is not only to maintain data consistency, as in conventional databases, but also to process transactions such that they satisfy their timing constraints.

1.1.1 Characteristics of a RTDBS

Typically, a real-time system consists of a *controlling system* and a *controlled system.* For example, in an automated chemical plant, the controlled system is the plant floor with its boilers, reactors, generators and piping; while the controlling system is the computer and human interfaces that manage and coordinate the activities on the plant floor. One can view the controlled system as the *environment* with which the controller interacts.

Data characteristics

The controlling system interacts with its environment based on the data available about the environment, say from various sensors, e.g., temperature and pressure sensors. It is *extremely important* that the state of the environment, as observed by the controller, be consistent with the actual state of the environment. Otherwise, the controller's actions might have disastrous consequences on the environment. Therefore, the timely monitoring and processing of sensed data is necessary.

The raw data obtained from the sensors is usually processed to derive further data. For example, the temperature and pressure data may be used to derive the rate of a particular chemical reaction. This processing is done within the controller. The raw sensor data often undergoes multiple levels of processing to generate control information which is used to directly activate the controller's

actuators. All data obtained by processing of raw data are called *derived data.* This is where another type of timing constraint comes into play. Note that if a process controller does not activate a pressure valve in a chemical boiler in time, the boiler might explode. Thus, the controller also has to comply with certain *response time* constraints.

The RTDBS has to maintain two types of temporal consistency in its data :

- **Absolute Consistency :** This requires consistency between the state of the environment and the state of the same as perceived by the controller in its database.
- **Relative Consistency :** This requires consistency in the derived data. This arises from the need to ensure that the derived data obtained from raw sensor data all correspond to the same (the latest) of the sensor data.

Transaction characteristics

Transactions in real-time database systems can be classified in three different ways : the manner in which they access data, the nature of the timing constraints and the importance given to finishing a transaction by its deadline [60].

Transactions can be classified as:

- **Write-only transactions**, which write into each data item they access, i.e., they consist of only write operations;
- **Read-only transactions**, which consist only of read operations; and
- **Update transactions**, which consist of both read and write operations. These typically read some data, perform calculations based on the data read and store results using write operations.

The above classification can be used to choose the appropriate concurrency control scheme for the RTDBS.

Transactions can also be classified as those with *periodic* time constraints and those with *aperiodic* time constraints. An example of a periodic timing constraint would be as follows :

Sample boiler temperature *every 20 seconds.*

Here, 20 seconds is the periodic time constraint. Note that if the RTDBS fails to sample and store a particular temperature reading within 20 seconds, the data becomes useless, since a new temperature reading has now arrived. The RTDBS aborts the old transaction and tries to store the newly arrived data within the next 20 seconds. An example of an aperiodic timing constraint is as follows :

If pressure in boiler > MAX PRESSURE, open valve *within 5 seconds.*

In this case the system's reaction to the pressure increase must be completed within 5 seconds.

Transactions can also be distinguished based on the consequences of missing a transaction's deadline.

- **Hard deadline transactions** are those which may result in a catastrophe if the transaction deadline is missed. These typically correspond to safety-critical activities such as emergency shutdowns and weapon systems. Another way of characterizing hard deadline transactions is by saying that a large negative value is imparted to the objective function of the system if such a deadline is missed.

- **Soft deadline transactions** are those which do not result in a catastrophe if the deadline is missed. Therefore, no negative value or cost is imparted to the objective function if a soft deadline transaction misses its

deadline. Instead, if the transaction completes execution within a certain time interval past its deadline, there is some positive value added to the objective function. Examples of soft deadlines are deadlines associated with components of a transaction. Note that if a transaction component fails to meet its deadline, the transaction, as a whole, may still be able to complete before the overall deadline.

- **Firm deadline transactions** are a special case of soft deadline transactions. Like soft deadline transactions, these do not result in a catastrophe and no negative value is imparted to the objective function in the event of a firm deadline miss. However, there is no positive value added if the transaction commits after its deadline has expired. For example, a transaction which brings in data periodically from a sensor has a firm deadline because if it does not complete before the next sensor report, the data becomes useless.

Thus, the three types of transactions all impart a positive value to the objective function of the system if they commit within their deadlines. However, they differ in the values they impart to the objective function if they miss their deadlines. A hard deadline transaction miss imparts a negative value to the objective function, a firm deadline transaction miss does not impart any value and, lastly, a soft deadline transaction miss may still result in a small positive value if it commits within a specific time interval past its deadline. The RTDBS scheduling algorithms therefore try to maximize the objective function by trying to complete as many transactions as possible and trying to minimize transaction misses, especially hard deadline transaction misses. The reader should also note that schedulers usually abort firm deadline transactions if they miss their dead-

lines. This is because their is no point in allocating resources to a transaction which will not impart any value (either negative or positive) to the objective function.

1.1.2 Transaction processing in a RTDBS

In this section we will discuss various aspects of transaction processing where the transactions have the characteristics described in the previous section.

Predictability

In a database system, it is almost never possible to compute exactly how long a transaction will take to complete its execution. This is because of a number of sources of unpredictability in a database system, some of which are listed below.

- Data and resource (CPU, disk etc.) conflicts
- Dynamic paging and I/O
- Transaction aborts and the resulting rollbacks and restarts

Since the consequences are catastrophic when a hard real-time transaction misses its deadline, one would like to be able to *predict* beforehand that such transactions will or will not be able to complete execution in time. The inherent sources of unpredictability just mentioned make this prediction very difficult. This problem is often overcome by computing the *worst-case execution times* for such transactions. The worst-case execution time, say W, is then compared with the transaction's deadline when it enters the RTDBS. If W is greater than the deadline, the transaction is immediately aborted and the system initiates emergency procedures. On the other hand, if W is less than the deadline, the system

can make use of the difference between the two times (called the *slack*) while scheduling the transaction operations along with those of other transactions.

Transactions with hard deadlines

Any transaction which has a hard deadline must meet its time constraints. To be able to ensure apriori that such a transaction will complete its execution in time, we need to guarantee the availability of resources and data whenever necessary. This in turn requires that we know the resource requirements and worst-case execution time of the transaction. This places many restrictions on the structure and characteristics of hard real-time transactions. However, since real-time systems are often developed as dedicated systems for highly customized applications, obtaining semantic knowledge about transactions beforehand is not uncommon. This knowledge often enables us to characterize transactions in the manner described above.

Once the transactions have been characterized based on resource requirements and worst-case execution times, it is possible to schedule them by using *static table-driven* or *pre-emptive priority-driven* scheduling schemes. In a static table-driven scheduling scheme, specific time slots are reserved for each transaction. If a transaction does not use all of the time reserved in its slot, the residual time may be used by the scheduler to start another hard-deadline transaction earlier than planned [67]. Other ways of dealing with the residual time are to schedule a soft-deadline transaction or simply remain idle. This approach is not very flexible. A priority-driven scheduler assigns priorities to transactions based on some priority assignment scheme. One example of a priority-driven approach is the *rate-monotonic priority assignment* scheme. In this scheme a

transaction's priority increases monotonically as it nears completion. This approach is discussed in [64], where periodic transactions accessing a main memory resident database are scheduled using rate monotonic priority assignment. *Schedulability analysis tools* [75, 76, 17, 2] have been suggested for such priority assignment schemes to check whether a set of transactions are schedulable given their deadlines and resource requirements. The schedulability analysis on a set of transactions is done with respect to the worst-case computational needs of the transactions.

If the variance between worst-case needs and the actual needs of the transactions is large, the scheduler will yield overly conservative schedules. Transactions whose average computation time requirements are much smaller than their worst-case computation time requirements will be pronounced infeasible by the scheduler since it considers their worst-case needs. Thus, we see that scheduling hard-deadline transactions is quite complicated. First, it requires placing many restrictions on the transactions themselves so that their characteristics are known apriori. Even if these restrictions are met, a system could still produce schedules with poor resource utilization if the worst-case assumptions about the transactions stray considerably from the average computation time requirements.

Transactions with soft deadlines

While scheduling soft-deadline transactions, the scheduler is not required to ensure that *all* transactions meet their deadlines. However, this does not mean that the scheduler simply lets a transaction run and aborts it if the deadline expires before it commits. Instead, the scheduler actively pursues the goal of

maximizing the percentage of transactions which complete their execution before their deadline expires. To achieve this goal, a scheduler uses various priority-assignment policies and conflict resolution mechanisms that explicitly take time into account. Priorities are used in scheduling transaction operations for the use of resources such as CPUs and disks. Conflict resolution schemes are used to resolve data contention among transactions. These schemes often use the transaction priorities to resolve these conflicts.

Many priority assignment policies have been proposed and extensively analyzed in the real-time database literature [55, 30, 65, 11]. Priority assignment is based on transaction deadlines and their importance or value to the system. Possible scheduling policies include :

- Earliest-deadline-first
- Least-slack-first
- Longest-executed-transaction-first
- Highest-value-first

1.1.3 Concurrency Control

Conflict resolution policies include various *time-cognizant* extensions of traditional concurrency control schemes such as two-phase locking, optimistic and time-stamp based protocols [2, 1, 19, 28, 27]. Some of these are discussed below.

Protocols based on two phase locking resolve lock conflicts using the timing information about the transactions. For example, [29] investigated the following protocols :

- If a transaction A requests a data item already locked by another transaction B with lower priority, then B is aborted. If A has lower priority, it waits for B to release the lock on the data item.

- If a lock-holding transaction is *close* to its deadline, then the lock-requesting transaction is forced to wait regardless of its priority.

- When a high priority transaction is forced to wait for a low priority lock-holding transaction, the low priority transaction is in fact taking precedence over the high priority transaction. Hence, this situation is called *priority inversion.* This is obviously an undesirable situation. One of the approaches to solving this problem involves *priority inheritance* where the lock-holding transaction *inherits* the priority of the lock-requesting transaction. In this way the lock-holder is made to finish sooner than with its own priority, which in turn allows the lock-requesting transaction to proceed more quickly.

In the *priority inheritance* scheme, the blocking time of the high priority transaction is reduced since the lock-holding transaction finishes sooner by executing at a higher priority. However, in the worst case, the high priority transaction still has to wait for the duration of a whole transaction. Therefore, the priority inheritance protocol typically performs worse than a protocol that always makes a lock-requesting transaction wait, irrespective of whether it has a higher or lower priority than the lock-holding transaction.

If a high priority transaction always causes a low priority transaction to be aborted, the low priority transaction may never complete execution due to countless aborts and restarts. This is especially true when there is high data

contention among the transactions. One solution to avoid this problem is the following. If the low priority transaction is close to completion, then it inherits the higher priority of the lock-requesting transaction. This considerably improves the performance under high data contention conditions. This protocol is essentially a combination of abort-based protocols used in traditional database systems [70] and the priority inheritance protocol proposed for real-time systems. Thus, even though techniques used in traditional database systems and those used in real-time systems are not applicable directly, they can often be tailored and combined to suit the needs of RTDBS.

Let us now consider optimistic protocols. *Backward validating* protocols allow a transaction to run freely until it reaches the commit stage. At this point the transaction enters the *validation phase.* In the validation phase, the transaction commits if it does not have any conflicts with other transactions which have already committed. If conflicts do exist, the validating transaction is aborted. This protocol has a disadvantage in that it does not take into consideration the transaction characteristics. *Forward validating* protocols do not have this disadvantage. Here, a committing transaction usually aborts any ongoing transaction which conflicts with the validating transaction. However, we do have the flexibility here to decide not to commit the validating transaction depending on its characteristics and those of the conflicting transactions. Several such policies have been studied in the literature [19, 18, 28].

In timestamp-based protocols, when data are accessed out of timestamp order, the conflicts are resolved based on the transaction priorities. In addition to the protocols discussed above, several combinations of lock-based, optimistic and timestamp-based protocols have been proposed [42].

Multi-versioning of data for enhanced performance has been investigated in [30]. Multiple versions of data reduce conflicts over data. However, they also introduce complications. Since data in RTDBS are required to have temporal validity, old versions have to discarded. Also, relative consistency must be maintained while accessing versions of related data.

This concludes the broad overview of some of the main research issues in RTDBS. One should, however, keep in mind the fact that RTDBS are a relatively recent concept and the research concerning issues pertaining to these systems is still far from mature. The main focus of RTDBS research so far has been the development and performance evaluation of scheduling algorithms. Once an algorithm is developed, it is evaluated primarily by simulation [74]. Future research may indeed include providing analytical frameworks for the study of such algorithms. These frameworks can be drawn from other areas which can effectively model RTDBS transactions, data and resources. It is with this line of thought we proceed to present a brief overview of discrete event dynamic systems (DEDS, or simply, DES). We will show that this theory provides a nice framework for studying various RTDBS transaction management systems.

1.2 Discrete Event Systems

A discrete event system (DES) evolves dynamically in time depending on the occurrence of various discrete events. For example, consider a manufacturing plant. Such a system evolves with the occurrence of discrete events such as arrival of a batch of jobs, completion of machining, breakdown of a machine etc. If we examine the state trajectory of a DES, we see that it is piece-wise constant

and changes only at discrete instants of time at which the events occur. Let us draw the state trajectory for our DES example, the manufacturing plant. For simplicity, let us assume that this plant consists of only one machine which can be in one of three states - *idle, busy* and *failed.* The events which cause the state changes are *arrival* of a job to be processed, *departure* of a machined part and *breakdown* of the machine. Figure 1.1 shows the state trajectory this system.

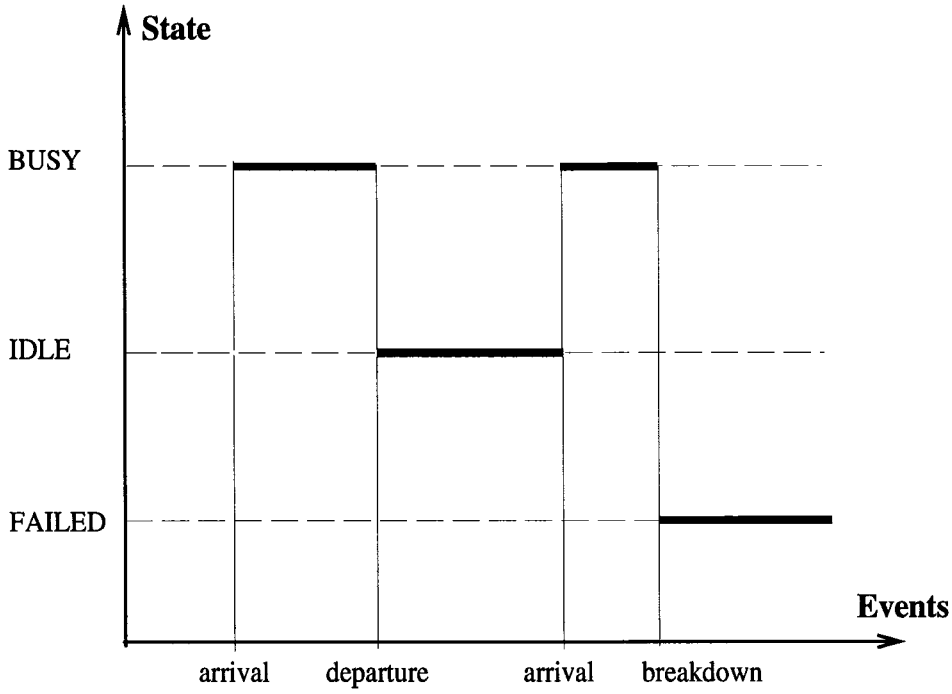


Figure 1.1: A DES state trajectory example

1.2.1 Modeling of DESs

A number of models have been suggested for DESs. Broadly speaking, these can be classified as follows :

- **Logical models** : These models are only concerned with the state-event sequences. The time spent at each state or the *state holding times* are

ignored. Examples of these models are finite state automata [26] and Petri nets [53].

- **Timed Models** : These models incorporate state holding times into the state specification. Examples are temporal logic models [14, 44, 54, 33] and timed Petri nets.

- **Performance Models** : These are stochastic models[1] used to evaluate various performance parameters of the system, e.g., throughput and delay in a communication network. These models are markov chains, queuing networks, GSMPs (generalized semi-markov processes) and simulation models.

We will concentrate mainly on logical models since that is what we have used in our modeling of the RTDBS. More specifically, logical untimed models such as Petri nets and finite automata are the most common models. These models make automated analysis of DESs relatively easy. The DES model for a RTDBS developed here involves the use of deterministic finite automata[2]. The timing information such as transaction deadlines is handled outside of the DES model. However, it is possible to extend this work by incorporating the timing information also into the DES model. This would require using timed DES models such as the one proposed in [5] which incorporate ticks of the clock in the event alphabet[3].

[1]Note that some of the logical models such as non-deterministic finite automata and Petri nets can also represent stochastic behavior.

[2]Given a current state and transition, one can uniquely determine the resulting state.

[3]The event alphabet is the set of events defined for the DES.

1.2.2 Applications of DES Theory

Most modern man-made systems are discrete, asynchronous and event-driven in nature. This has brought into prominence the study of DESs in recent years. The efficient operation of these increasingly sophisticated and complex systems has prompted the use of various DES analysis, modeling and control tools in diverse application areas such as flexible manufacturing systems [37, 16, 41, 6, 25], telecommunication systems [61, 51], semiconductor chip manufacturing [22, 3], parallel processing [24] and database management systems [34, 32, 63].

Real-time systems such as weapon systems and avionics, air traffic control systems etc., where the issue of timing is critical, has become one of the most active areas of research in DES control theory. [47, 48, 8, 46, 49] initiated the work in this area and further developments were proposed in [7, 23, 50, 51].

1.3 Motivation, Objectives and Contribution

Broadly stated, this research is an attempt at synthesizing ideas between two disciplines that appear not to have gained much from each other in the past, namely database systems and DES. The motivation is the belief that they have much to gain from this synthesis. More specifically, real time database systems appears to be an excellent area to apply ideas in DES, particularly in the context of scheduling and concurrency control. We make such a claim based on the natural fashion in which databases systems in general, and RTDBSs in particular, fit the DES framework. Our DES model is presented in section 2.2.

Another motivation for using DES models to study RTDBS is the considerable amount of work done by the computer science community in develop-

ing frameworks for modeling, specification, verification and synthesis of discrete event processes such as computer operating systems, concurrent programs, distributed processes and database management. Various approaches developed in this area are Petri net theory [53], linear-time and branching-time temporal logics [14, 44, 54, 33], concurrent process algebras such as Hoare's communicating sequential processes [21] and Milner's calculus of communicating systems [45]. The last two inspired the development of a number of algebras of concurrent processes which became widely known as the theory of concurrency [10, 9, 20]. An important aspect of this theory deals with the interaction between DESs and their environment. Such interaction is modeled by parallel composition with a specified degree of event synchronization. Various forms of parallel compositions have been defined and investigated in the literature of concurrency theory. Such compositions appear (and prove to be) very useful in modeling transaction execution in RTDBSs. More specifically, we perceive transaction execution as transactions accessing data items (resources), where the transactions and resources (data items) are modeled as discrete event systems (DESs) and the composition methods mentioned above are used to model the interaction between the events in the transaction and resource DESs. In this thesis, we use automata theory [26] to model transaction and resource DESs as deterministic finite automata or DFAs. This is because of the fact that finite automata models make automated analysis relatively easy[4].

Another important goal of this work is to motivate research in the synthesis of supervisory control theory and RTDBSs. Note that the main body of work

[4]This is despite the fact that state space explosions make some problems formulated as automata interactions intractable.

presented in this thesis mainly deals with finite automata theory to represent and manipulate DESs. However, the objective of this research is to eventually apply supervisory control theory to the RTDBS area. As we show in Chapter 4, our approach can be viewed as a special case of the supervisory control approach in which all events are controllable.

Supervisory Control

In the framework of automata and formal languages proposed in [58], the supervisory control theory of discrete event systems has very successfully treated a variety of abstract synchronization problems defined by specifications of a qualitative or "logic based" type. These include "safety" specifications (e.g., service priorities, exclusion from prohibited states) and "liveness" properties (e.g., guaranteed eventual entrance into a goal state). An excellent review of the ideas behind supervisory control of DES is available in [59]. Real time database systems fit very naturally in this framework. Intuitively, it may be seen that in real time scheduling the basic objective is the control of transaction execution in order to satisfy certain constraints or *specifications*. Formally it has been shown that transaction execution can be modeled and analyzed as a discrete event dynamical system [34]. Such analysis may allow researchers to prove new results about the performance of existing concurrency control (CC) techniques such as locking [15] or timestamp ordering [4] as well as new ones that may be developed. A similar case may be made for scheduling algorithms. In particular, an example of such analysis is offered in [34] where the author shows that schedulers corresponding to the CC schemes mentioned above may be embedded into an ideal "complete information" scheduler whose state space is a set of graphs. This embedding is

then used for analytical comparison of the performance of such schedulers.

The objectives of this work may therefore be summarized as follows :

- To make a convincing case for the integration of real-time databases and discrete event dynamical systems theory.
- To motivate the application of supervisory control theory of discrete event systems in the analysis of RTDBS.

The domain of concurrency control and scheduling [74, 31, 43, 17, 36, 52] is one of the most active areas of research in RTDBS. The common theme of research in this context is the stipulation and subsequent performance evaluation of CC and scheduling policies. Thus after a new model has been postulated, a large effort is consumed in analyzing and validating it. It is in such analysis and validation that we wish to contribute in this work. However, the purpose of this research is not to postulate new scheduling or CC mechanisms. Rather, we propose an alternative but elegant way to model and evaluate such mechanisms using notions from the theory of DEDS represented as finite state automata.

The contributions of this thesis are summarized as follows :

- We have developed a discrete event dynamical system model of transaction processing in a RTDBS using *deterministic finite automata* (DFA).
- An algorithm for on-line concurrency control and scheduling has been developed using this DES model.
- We have developed a simulator, RTSIM, to simulate CC and scheduling in RTDBS using various CC and scheduling policies.

1.4 Thesis Outline

The rest of this thesis is organized as follows. Chapter 2 describes our framework for the analysis and evaluation of RTDBS in detail. In Chapter 3, we describe the simulation software RTSIM and the simulation of a particular RTDBS configuration using RTSIM. In Chapter 4, we introduce supervisory control theory of DES and argue why this theory promises to be a powerful tool in RTDBS research. Conclusions are provided in Chapter 5. Appendices A and B illustrate a "toy" example of the application of the theory developed in section 2.2.

Chapter 2

The RTDBS Model

2.1 Architecture of a RTDBS

We consider an RTDBS architecture as shown in Figure 2.1. This RTDBS is a centralized database system and is similar to the database system model in [4] and [63]. It consists of one or more CPUs, some main memory, secondary storage devices (disks) and I/O devices.

The system shown consists of five modules :

- a **transaction manager (TM)**, which processes incoming transactions and prepares them for scheduling;
- a **data manager (DM)**, which processes the individual operations of the various active[1] transactions and operates directly on the database to implement the transaction commits and aborts;
- the **database** itself, which is a set of data items (files/records/fields/pages);

[1]Note that we use *active* in the sense of transactions that are being currently processed and not in the sense of triggered transactions.

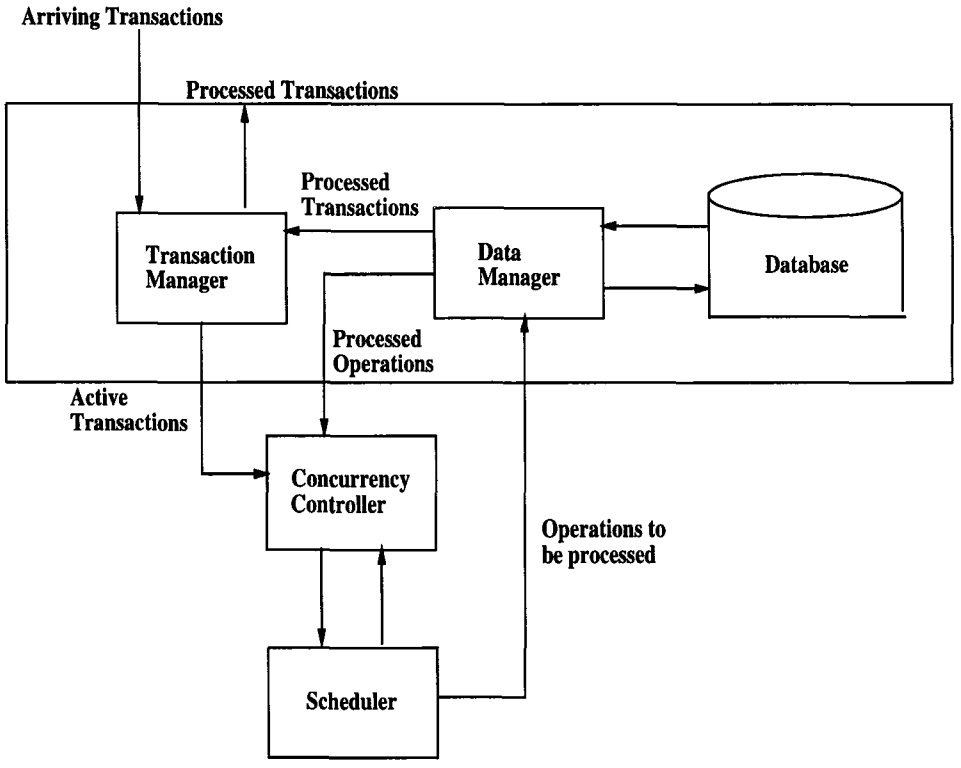


Figure 2.1: Real-time Database System Architecture

- a **concurrency controller (CCR)** which maintains database consistency, and;
- a **scheduler** that schedules transactions to meet deadlines.

Our basic approach is as follows: The CCR employs a particular concurrency control algorithm to generate all possible *legal* interleavings[2] (which we call *traces*) of transaction operations; these traces are then forwarded to the scheduler, which selects one particular sequence of interleavings or trace from the set of traces generated by the CCR based on a particular scheduling policy.

[2]Legal interleavings are those interleavings which meet the requirements of the CC algorithm being employed.

Transactions arrive for processing with deadlines. When a transaction enters the system, it first passes through a buffer called the Ready Queue which is used to store arriving transactions when the system is already full (i.e., number of transactions in the system is equal to the *multiprogramming level* (mpl)). When the system is ready to process a new transaction, the TM assigns it a unique *transaction id.* and forwards it to the CCR.

The CCR models the database and transactions as a discrete event system and performs computations to generate legal interleavings or traces using the DES model of the database. Most RTDBS scheduling formulations integrate concurrency control within the scheduler [63]. In our model, we have chosen to create separate modules for concurrency control (CC) and scheduling. There are two reasons for this separation. First, the CCR module is implemented as a composition of the transaction and resource DES automata (which we call DFAs) which is synthesized on-line as a solution to a discrete event system (DES) problem. It does not deal with any timing information such as transaction deadlines and hence represents the consistency maintenance part of the RTDBS. Thus, the CCR module allows us to clearly distinguish between the consistency maintenance part of the scheduling problem and the scheduling policy implementation itself which utilizes all the timing information. The second, and more important reason is that this framework provides the flexibility of experimenting with different combinations of concurrency algorithms and scheduling policies in a modular fashion. For example, we could model systems with concurrency control-scheduling policy combinations such as (2pl, earliest deadline), (2pl, least slack) etc. and compare their performance through simulation.

After the CCR generates legal traces, the scheduler is invoked. The scheduler

now processes this set of traces and picks one based on some scheduling policy (e.g., *earliest-deadline-first*). More precisely, the scheduler computes priorities of active transactions based on the scheduling policy chosen. Then it picks the trace which most closely fits its priority order. Therefore, in the selected trace, the operations of high priority transactions occur at the beginning and the operations of low priority transactions appear at the end. The trace selection process is described in detail in section 2.3. Sometimes, it may happen that none of the traces passed to the scheduler fit its priority order. For example, none of the traces may execute an operation of the highest priority transaction at the beginning (e.g., the first five operations in that trace). This signifies a scenario when the scheduler cannot find a feasible schedule without preemption. In such a situation, the scheduler can abort (i.e., preempt) a particular transaction. This abort is communicated to the CCR, which recomputes a supervisor based on the modified transaction set and then sends a fresh set of legal traces to the scheduler. The trace chosen by the scheduler is passed to the DM which then proceeds to execute the operations in the trace one by one. The DM also communicates to the CCR so that the latter can dynamically update its DES model of the data items and transactions. If a transaction misses its deadline, it is aborted by the scheduler.

2.2 A DES model of the RTDBS

As mentioned in section 2.1, the CCR module operates on a DES model of the RTDBS to generate legal traces. In this section, we describe this DES model in detail. The resources (data items) in the database and the transactions accessing

them are modeled as deterministic finite automata (DFAs)[3].

2.2.1 DES operators

Before describing the DES models for the transactions and the resources let us first look at two commonly used DES operators. These operators are used to model the interaction between two DESs. We will use these operators to model the interaction between transactions and resources.

Shuffle product

Consider two DESs, T and R, operating independently and asynchronously. Figure 2.2 shows the discrete finite automata (DFA) representations of T and R. The circles represent *states*, the arrow labels represent *events* and the arrows themselves represent state transitions corresponding to these events. The states represented with two concentric circles represent *marked* (goal) states. We model

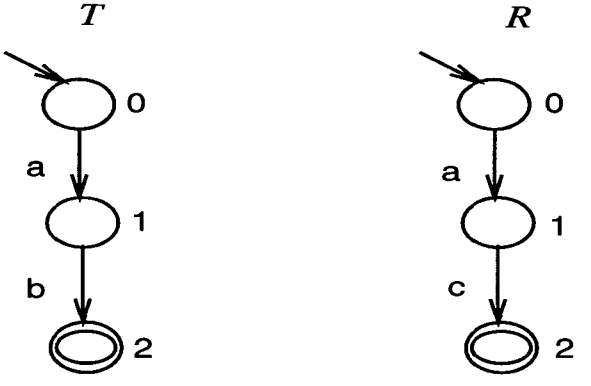


Figure 2.2: DFA representation of two DESs

the joint operation of these two DESs by the shuffle product, $T||R$, where ||

[3]DFAs are a convenient way of representing Discrete Event Systems.

denotes the shuffle operator. The states of $T||R$ are ordered pairs (X, Y) where X is a state of T and Y is a state of R. The transitions of $T||R$ are either of the form $(X, Y) \rightarrow (X', Y)$ where $X \rightarrow X'$ is a transition in T, or of the form $(X, Y) \rightarrow (X, Y')$ where $Y \rightarrow Y'$ is a transition in R. The DFA for $T||R$ is shown in Figure 2.3. Informally, we say that the operations of T and R are interleaved in an arbitrary manner since their actions are asynchronous and independent of each other. Note that the marked states in the shuffle product are simply the

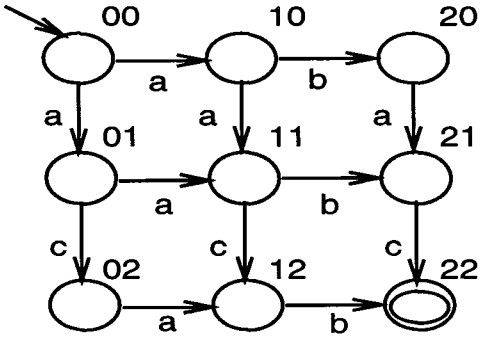


Figure 2.3: Shuffle product of T and R

cartesian product of the marked states in T and R, i.e., a state (X, Y) in $T||R$ is marked only if both X and Y are marked in T and R respectively. In this case, we have only one marked state labeled as 22 in Figure 2.3.

Synchronous composition

Now assume that T and R are no longer operating indepently of each other. Instead, they have to synchronize on some common activities and events. In our example, this common event is a. This kind of interaction is modeled by another

operator called synchronous composition[4] (SC) denoted by $\square$. The states of $T \square R$ are also orderd pairs (X, Y) where X and Y are states in T and R respectively. The transitions in $T \square R$ consist of those described in the shuffle product, i.e., $(X, Y) \rightarrow (X', Y)$ and $(X, Y) \rightarrow (X, Y')$ when the corresponding transition event label is present in only one of T and R and not in both. However, there is an additional type of transition of the form $(X, Y) \rightarrow (X', Y')$ where $X \rightarrow X'$ is a transition in T, $Y \rightarrow Y'$ is a transition in R and both these transitions have the same event label. See Figure 2.4 for the synchronous composition of T and R. The marked states in $T \square R$ are simply the cartesian product of the marked states in T and R.

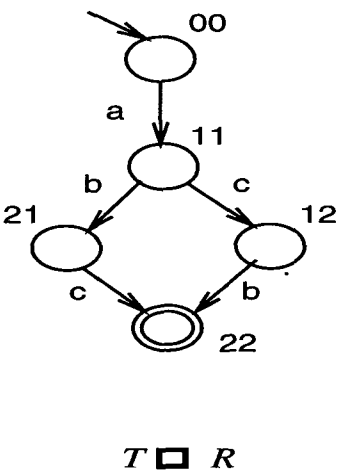


Figure 2.4: Synchronous composition of T and R

Informally, an event σ present in the alphabets of two DFAs, D_1 and D_2, can occur in $D_1 \square D_2$ only if both D_1 and D_2 allow σ in their current states. Thus, if some common event is not allowed in either D_1 or D_2, it will not be allowed in their SC. All other events (i.e., those which are not common) can be executed

[4][21] refers to SC as Full Synchronous Composition or FSC.

in an unrestricted fashion by the DFA in whose alphabet they appear.

2.2.2 Transaction Model

In this section we show the discrete event system (DES) model of a transaction. We first provide an intuitive description followed by a formal treatment. Consider a transaction $T_1 = read(1), write(2)$, where 1 and 2 represent data item (resource) ids.

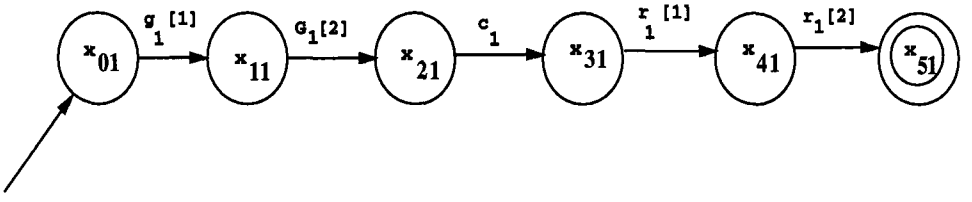


Figure 2.5: Transaction Model

Figure 2.5 shows the discrete finite automaton (DFA) corresponding to transaction T_1. This DFA is a representation of a DES model for transaction T_1. The transition labels contain transaction identifiers as subscripts and resource identifiers in square brackets. For example, $G_1[2]$ denotes the exclusive grab (e.g., write operation on a data item) of resource 2 by transaction 1, 'c_1' stands for commit operation of transaction 1 and '$r_1[2]$' means release of resource 2 by transaction 1. The release events need special mention as they do not usually appear in database literature as operations of a transactions. However, we need to explicitly model release events as they form the basis of resource sharing.

After a transaction finishes execution (i.e., the grabbed resources are processed) it performs a commit operation to make its updates (if any) permanent and subsequently relinquishes its resources. Note that this model represents the

2-phase locking protocol since the resources grabbed by the transaction are held until commit time.

We formally model a transaction T_i as a 5-tuple:

$$T_i = (X_i, \Sigma_i, \delta_i, x_{0i}, X_{mi})$$

where

- X_i is the state space of T_i. For example, in Figure 2.5,

$$X_1 = \{x_{01}, x_{11}, x_{21}, x_{31}, x_{41}, x_{51}\}$$

- Σ_i is the event alphabet. In Figure 2.5,

$$\Sigma_1 = \{g_1[1], G_1[2], c_1, r_1[1], r_1[2]\}$$

- x_{0i} is the initial state. In Figure 2.5 this corresponds to the state x_{01}.

- X_{mi} is the set of marked (goal) states (i.e., states signifying the completion of T_1). In our case, transactions will typically have just one marked state as shown in Figure 2.5.

$$X_{m1} = \{x_{51}\}$$

- $\delta_i : \Sigma_i \times X_i \mapsto X_i$ is the (partial) state transition function. In Figure 2.5, the state transition function of T_1 is defined as follows: $\delta_1 : \Sigma_1 \times X_1 \mapsto X_1$

$$\begin{aligned}
\delta_1(g_1[1], x_{01}) &= x_{11} \\
\delta_1(G_1[2], x_{11}) &= x_{21} \\
\delta_1(c_1, x_{21}) &= x_{31} \\
\delta_1(r_1[1], x_{31}) &= x_{41} \\
\delta_1(r_1[2], x_{41}) &= x_{51}
\end{aligned}$$

δ_1 is undefined for all other (σ, x) pairs where $\sigma \in \Sigma_1$, $x \in X_1$

Each active transaction T_i in the RTDBS is modeled as a 5-tuple just like the one described above. However, the reader should note that the various components of the 5-tuple (i.e., event alphabet, state space, transition function etc.) will vary from transaction to transaction depending on the transaction id., and the number and types of operations (shared/exclusive grab) in the transaction. However, all transactions have the same basic structure, i.e., *each transaction is a linear order of grab, commit and release events.*

The event alphabets of the transactions will clearly be disjoint because the event labels use transaction ids as subscripts and each active transaction is assigned a unique identification number. Hence, we can model the concurrent operation of these transactions by simply interleaving the transitions of all the transaction DFAs. Therefore, we compute the shuffle product of the transaction DFAs. The resultant DFA, T, is thus computed as follows :

$T = T_1 \,||\, \ldots \,||T_N$

where N is the number of active transactions and $||$ denotes the shuffle operator. We will represent T also as a 5-tuple :

$$T = (X, \Sigma, \delta, x_0, X_m)$$

where

- $X = X_1 \times \ldots \times X_N$
- $\Sigma = \Sigma_1 \cup \ldots \cup \Sigma_N$
- $x_0 = (x_{01}, \ldots, x_{0N})$
- $X_m = \{(x_1, \ldots, x_N) \mid x_1 \in X_{m1}, \ldots, x_N \in X_{mN}\}$

- $\delta : \Sigma \times X \mapsto X$ is the transition function,

  $\delta(\sigma_k, (x_{i_1 1}, \ldots, x_{i_k k}, \ldots, x_{i_N N})) = (x_{i_1 1}, \ldots, \delta_k(\sigma_k, x_{i_k k}), x_{i_N N})$

  where $\sigma_k \in \Sigma_k$, $\delta_k(\sigma_k, x_{i_k k})!$ and $1 \leq k \leq N$ and the symbol ! is read as "is defined".

2.2.3 Resource Model

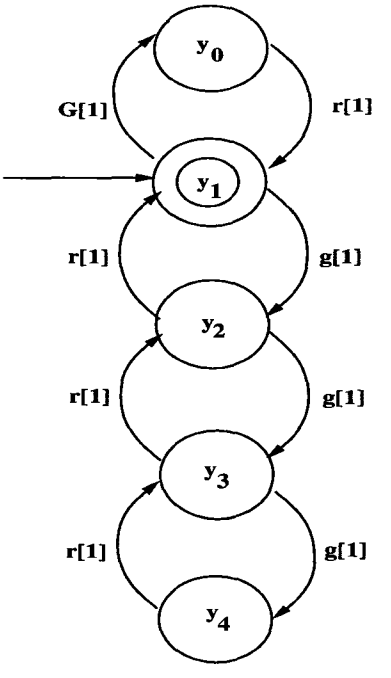


Figure 2.6: Resource Model

Figure 2.6 shows a resource DFA for a resource R_1. The transition labels $g[1]$, $G[1]$ and $r[1]$ correspond to the shared grab (read), exclusive grab (write) and release of the resource where the index 1 indicates *resource id.* Thus, Figure 2.6 expresses the facts that the resource may be exclusively grabbed by one transaction, but may be grabbed in a shared fashion by a maximum of three transactions. Typically this number would be equal to the *mpl* of the system. Also, note

the absence of any transaction identifiers in the labels. There are two reasons for this :

- To make the resource model independent of the number and type of transactions - both of which vary with time; and
- Dropping the transaction identifiers from the resource DFA transition labels also reduces the state space of the resources considerably. If we had chosen to keep the transaction identifiers we would have had to add states for all possible combinations of transactions grabbing the various resources (e.g., with two transactions in the system, there would be 16 possible states).

The second reason alleviates, to a certain degree, the problem of state space explosion in automata composition. Note however, that this makes the transaction and resource DFA alphabets disjoint (as there are no common event labels) which leads to complications in synchronizing them. The solution to this problem is offered in section 2.2.5.

Similar to the transaction DFA, a resource DFA like the one shown in Figure 2.6 represents a resource DES which may be formally represented as (for the resource R_1) :

$$R_1 = (Y_1, \Sigma_1', \alpha_1, y_{01}, Y_{m1})$$

where the symbols have their usual meaning as shown in the transaction example.

The composite resource model is given by the *shuffle product* of DFAs like the one shown in Figure 2.6, one for each resource. The resultant DFA, R, is thus given by :

$$R = R_1 \| \ldots \| R_J$$

where J is the number of resources in the database. For an example of resource shuffle product computation, see appendix A, particularly Figures A.1B, A.1C and A.3B.

Again, we represent R also as another 5-tuple :

$$R = (Y, \Sigma', \alpha, y_0, Y_m)$$

where each component of the tuple is obtained by applying the shuffle product definition (as shown for T).

2.2.4 Transaction Aborts

The reader may have noticed the absence of any *abort* event label in the transaction DFA model. This is because the abort event transitions need not be explicitly modeled, but are issued by the scheduler. However, aborts do form an important component of our system. Basically, there are two ways in which an abort may be handled in our model:

- **Preemptive aborts :** If an aborted transaction is to be restarted (i.e, a case of preemption), the CCR can simply reset the current state pointer of the transaction DFA to the initial state of the DFA (thus simulating an arrival of the same transaction) and fire release events in the resource DFAs which correspond to the resources locked by the particular transaction at abort time (thus freeing up the resources); and

- **Non-preemptive aborts :** If the aborted transaction is to be discarded, as is the case when a transaction misses its deadline, the CCR simply destroys the transaction DFA after firing release events in the relevant

resource DFAs. We will return to the handling of aborts by the scheduler in section 2.3.

2.2.5 Transaction and Resource Synchronization: The CCR Model

Transactions operating concurrently in an unrestricted manner (as modeled by T) do not yield legal schedules or traces - there has to be some agent which can enforce concurrency constraints (e.g., serializability) based on a knowledge of conflict resolution strategies (e.g., lock compatibilities in lock based protocols). The lock compatibilities are indeed represented in our resource model. If we make the resource model R interact with our transaction model T, we can force the transaction to obey the locking constraints. This kind of interaction is modeled by synchronous composition (SC).

The basic idea behind our approach towards concurrency control is to restrict the sequence of operations (or traces) to one that satisfies the CC policy being considered. Considering T as a spontaneous generator of events, we can achieve the above goal through synchronizing T with another DFA, say V, thereby yielding a resultant DFA, say C. V should be such that it will prevent illegal traces from occurring in C by not providing common event(s) required for synchronization at those points in the trace where the execution of such event(s) would result in an illegal schedule. One would be tempted to choose R itself as a candidate (i.e., as the DFA, V) for synchronization with T to obtain a DFA, C, which consists of only legal traces. However, $T \square R$ will not yield the desired result. This is because the two DFAs have disjoint alphabets (since we dropped the transaction identifiers from the transaction DFA event labels to get the cor-

responding resource DFA event labels). This means that T could still execute all operations in an arbitrary fashion without any locking constraints (since SC places the constraint of event synchronization only on common events). This motivated the creation of a new operator called the *masked synchronous composition* (MSC) which is used to modify R so that its event alphabet becomes the same as T. It is *this* modified R then, which interacts with T via SC to obtain the desired DFA with legal traces only.

Modifying Specification R using Masked Synchronous Composition

We will first explain the intuition behind MSC (as applied here). This is best done through an example. Consider a resource i which has been locked by a transaction j for a write operation. This corresponds to the transitions $G_j[i]$ in the transaction j DFA and $G[i]$ in the resource i DFA occurring *synchronously.* Similarly, when the transaction j releases the lock on resource i, the transitions $r_j[i]$ and $r[i]$, in the transaction and the resource DFAs respectively, synchronize to model this activity. Thus, we see that although we need to synchronize events with different labels in the transaction and resource DFAs, there is a well-defined relation between the two labels. In this case, the relation between the transaction and resource DFA event labels can be expressed as a simple projection function which projects out the transaction identifiers from the transaction event labels to yield the corresponding resource event labels. We call this function, the *masking function, M.* So, instead of synchronizing events with the same labels (like in SC), we now synchronize events related by the masking function M.

The MSC of T and R, denoted by $T \square^M R$, can now be formally defined as

follows:

$$T \square^M R = (Z, \Sigma_Z, \beta, z_0, Z_m)$$

where the symbols have their usual meaning and are defined as follows[5]:

- $Z = X \times Y$
- $\Sigma_Z = \Sigma_T$
- $z_0 = (x_0, y_0)$
- $Z_m = \{(x, y) \in Z \mid x \in X_m \wedge y \in Y_m\}$
- The partial state transition function β is defined as follows. Let $\sigma \in \Sigma_Z$ and $(x, y) \in X \times Y = Z$. Then

$$\beta(\sigma, (x, y)) = \begin{cases} (\delta(\sigma, x), \alpha(M(\sigma), y)) & \text{if } \delta(\sigma, x)! \text{ and } \alpha(M(\sigma), y)! \\ (\delta(\sigma, x), y) & \text{if } \delta(\sigma, x)! \text{ but } M(\sigma) \text{ is not defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

The masking function M is defined as follows[6]:

$M : \Sigma_T \mapsto \Sigma_R$,

$M(o_i[j]) = o[j]$, where $o \in \{g, G, r\}$

The reader will note that $M(c_i)$ is not defined. This means that T can execute commits without being constrained by R in their MSC. This makes sense intuitively, since R models only the grabbing and relinquishing of resources and therefore, need not be concerned with the commits of the transactions.

[5]The reader should refer back to the 5-tuple DES representations of T and R in order to understand this definition.

[6]M is defined as a simple projection function here. Clearly, in a different context, an MSC could be based on a more complex function.

We use the MSC of T and R to generate a modified specification which will have the same alphabet as T. In this manner, we modify our original specification R to obtain $T\square^M R$ whose marked language[7], say K, now corresponds to the set of legal traces. In other words,

$$L_m(T\square^M R) = K$$

We now use this modified specification, $T\square^M R$, to synchronize with T using SC. Thus, $T\square(T\square^M R)$ yields the DFA which represents all the valid schedules. This is the same as $T\square^M R$ itself because :

$L_m(T\square^M R) \subseteq L_m(T)$ and

$L_m(T\square(T\square^M R)) = L_m(T) \cap L_m(T\square^M R)$.

So we can compute the legal traces by simply computing the marked language of $T\square^M R$.

The Synthesis Algorithm

To summarize then, the Synthesis Algorithm, which is implemented by the CCR module to generate legal traces of transaction operations, consists of the following steps :

Whenever, a new transaction arrives,

- Create a DFA for the new transaction.
- Compute the shuffle product T, of all Transaction DFAs. This means shuffling the new DFA with the DFAs of already active transactions. The

[7]Marked language of a DFA is the set of all traces which lead to marked states from the initial state. See [26, 59] for a formal definition .

DFAs for the latter are modified so that their current states become their new initial states, before the shuffle is computed.

- Compute the shuffle product R, of all Resource DFAs in their current states.
- Obtain the MSC of T and R, i.e., $T \square^M R$.
- Obtain a list of all traces of the resultant DFA to get the list of valid concurrent schedules.

See Appendix A for a detailed example of the above computation procedure for synthesizing legal traces.

2.3 The Scheduler

The scheduler is the RTDBS module which is responsible for scheduling transactions in a manner such that they meet their deadlines. The information the scheduler has at its disposal are the deadlines of the active transactions. Whenever a new transaction arrives in the system, the CCR executes the Synthesis Algorithm outlined above and forwards the scheduler a set of legal traces. The scheduler then computes priorities of the various transactions based on some scheduling policy such as the *earliest-deadline-first* (EDF), to select a trace whose operations best fit this priority order. We have simulated an earliest-deadline-first scheduling policy using RTSIM, a simulator developed specifically for implementing the methodology developed in this work. A description of RTSIM and simulation results for the EDF scheduler are presented later in Chapter 3.

To understand exactly how the scheduler works we provide the following example. Suppose two transactions enter an initially empty RTDBS. Let them be assigned transaction ids 1 and 2. Also assume that T_1 and T_2 consist of the following sequences of operations:

$$T_1 = g_1[1]\, c_1; \text{ and}$$
$$T_2 = G_2[1]\, c_2$$

Further assume that the deadline for T_1 is d_1, and that for T_2 is d_2, where $d_1 < d_2$. The scheduler then assigns T_1 a priority of 1 and T_2, a priority of 2 (using EDF). It also obtains a list of legal schedules from the CCR module. In our example, this set would consist of the following traces: (a.) $g_1[1]\, c_1\, G_2[1]\, c_2$; and (b.) $G_2[1]\, c_2\, g_1[1]\, c_1$.

Since T_1 has higher priority, the first trace fits the priority order of transactions better. This is because T_1 operations are scheduled first in this trace. Therefore, the scheduler picks the 1st trace and starts scheduling the operations one by one, i.e., first $g_1[1]$, then c_1 and so on. It also keeps updating the current transaction deadlines as time passes. The database manager, DM, processes these individual operations and communicates with the CCR. The CCR updates its DES model accordingly. For example, after $g_1[1]$ is processed by the DM, the CCR fires the transitions $g_1[1]$ in the T_1 DFA and also $g[1]$ in the *resource 1* DFA. This process is repeated for each operation scheduled.

If a new transaction t_i arrives meanwhile, the CCR reruns the Synthesis Algorithm on-line. It computes the new transaction model T (T consists of the shuffle including an additional t_i now) and modified resource model R (note that the resources have changed states too). It then computes the MSC of the new T

and R and extracts all legal traces from the resultant DFA and passes them to the scheduler. The scheduler uses the new transaction's deadline along with the updated deadlines of the already active transactions to recalculate priorities and chooses a trace again based on the priority order calculated. This procedure is repeated for every transaction arrival.

Whenever a transaction t_i commits, the corresponding deadline information is discarded by the scheduler. The CCR destroys the corresponding transaction DFA and fires the release resource $r_i[j]$ for all resources j locked by t_i. If a transaction's deadline expires before it completes, the scheduler and CCR perform the same operations as in the commit event. However, the transaction is discarded by the system.

As mentioned in section 2.2.5, the scheduler sometimes might be faced with the situation that none of the traces passed to it by the CCR fit the priority order which it has computed for the active transactions. In this situation, the scheduler has to abort at least one transaction and communicate this to the CCR. The CCR then recomputes a new set of legal traces and passes them back to the scheduler. The scheduler now picks one trace from this set. If an abort is required again, the same process is repeated. The criterion for deciding to abort a transaction is based on the scheduling policy being implemented. In our simulation of the earliest-deadline-first policy, this decision would be taken when none of the traces have their first operation (to be scheduled) as a member of the priority one transaction. The scheduler then aborts one of the active transactions (the choice of which can be based on a number of criteria such as number of conflicts, number of remaining operations, transaction priority, random choice, etc.) and communicates this abort to the CCR. The CCR then

updates its DES model of the transactions T (removes the aborted transaction from the plant) and also the resource R (fires release events in relevant resources). It then recomputes the synchronization with the modified DFAs for R and T, extracts a new set of traces from the synchronization, and passes them to the scheduler. Meanwhile, the aborted transaction is restarted by the Transaction Manager after a small random time interval[8].

See appendix B for an example of scheduler operation.

[8]This is because immediate restarting would cause repeated aborts. The restart interval is a parameter which can be tuned.

Chapter 3

RTDBS Simulation and Results

To demonstrate the potential of the framework developed in the previous chapter as an evaluation tool of concurrency control-scheduler pairs, we implemented RTSIM (real-time simulator), a simulation software which can be used to simulate transaction execution in a RTDBS with any CCR-scheduler combination. Note that RTSIM is a "software package", as opposed to a simple simulation program, i.e., our software may be used to implement a variety of CC-Scheduling protocol combinations simply by changing the CCR and Scheduler modules, while leaving the remaining modules intact. In this chapter, we discuss the implementation of RTSIM and the results obtained from the simulation runs carried out for a particular CCR-scheduler pair, namely, 2pl-EDF.

3.1 Simulation Tools

The simulation software RTSIM was implemented using the SIMSCRIPT II.5 programming language [35] and ANSI C. SIMSCRIPT is a simulation modeling language with powerful Discrete-Event Simulation capabilities. It supports the

modeling concepts of *entities* and *processes* and is suitable for event-based and process-based simulation of discrete systems.

RTSIM is composed of a group of SIMSCRIPT routines which interact with a group of C programs. The SIMSCRIPT programs handle the actual dynamics of the RTDBS such as the arrival of transactions, assigning transaction ids, queuing operations of transactions in the CPU and DISK resource queues, etc. The DES model of the CCR and the various update procedures were implemented as a collection of C routines built on top of a finite state machine (FSM) library, also written in C. This library was originally developed in the University of Texas, Austin [62]. We have modified and extended it to incorporate several new functions required for this framework. The CCR programs create and update the DES models of the transactions and resources. The CCR also has C routines to compute the legal traces as illustrated in Appendix A.

RTSIM was designed in a highly modular fashion. The interfaces between the modules were made as simple and as general as possible. This enables us to simulate various CCR-scheduler combinations by simply replacing the CCR and scheduler modules with alternate modules. This way minimal additional coding and modification is required to test different RTDBS configurations.

3.2 The Simulator

Figure 3.1 shows a schematic of RTSIM identifying the various program modules. The following is a detailed description of the various modules and their functions. Pseudo-code for the various routines used by the different modules is also included. The reader should be aware that the modules shown in Figure 3.1

do not have a one-to-one correspondence with the SIMSCRIPT and C routines. Some of the modules consist of multiple routines while some routines are shared by more than one module. However, the CCR and Scheduler modules have been designed in a manner so as to make them independent of the rest of the module implementations.

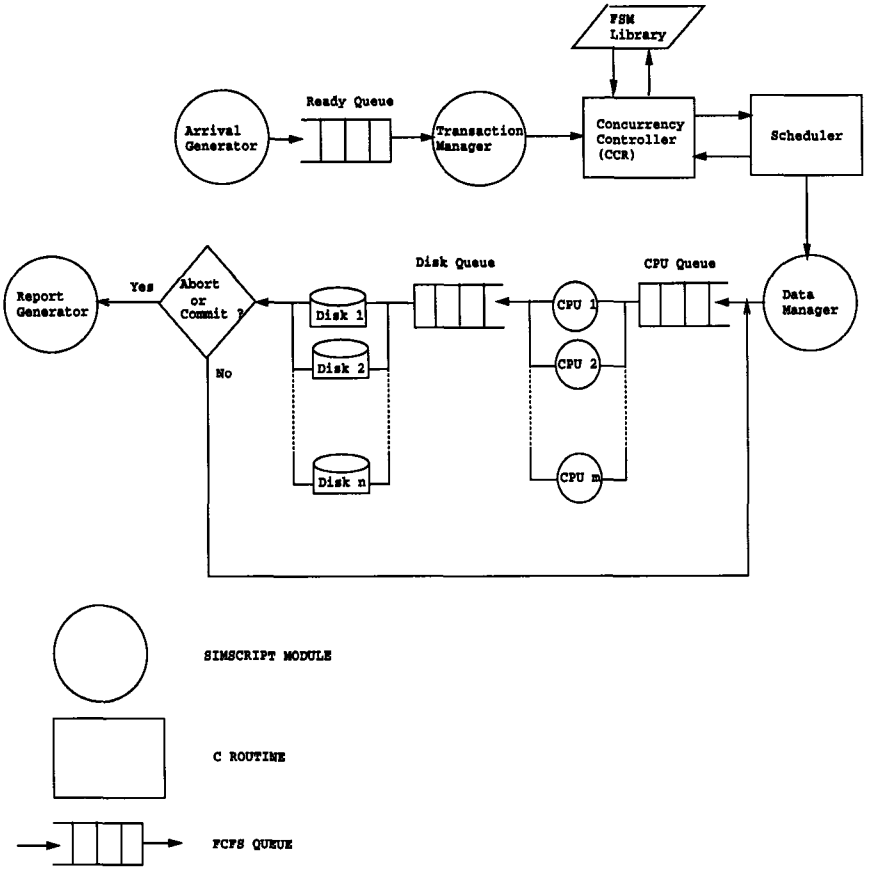


Figure 3.1: Simulation Software Block Diagram

3.2.1 Arrival Generator

The *arrival generator* module is a SIMSCRIPT process which generates new transactions with exponentially distributed interarrival times. Transactions en-

tering the RTDBS typically have different grab operations. The arrival generator models the arrival of different types of transactions by randomly generating a *transaction type.* Each *transaction type* corresponds to a specific sequence of grab, commit and release operations. The *transaction type* can take values from 1 to N, where N is the number of different types of transactions specified in the simulation. Since this is an RTDBS, each transaction enters the system with a deadline d. d is a function of the *transaction type*[1] and a parameter called the *slack ratio.* The slack ratio may be informally considered a "difficulty of schedulability" parameter. The deadline d assigned to a transaction is simply the product of the slack ratio and the computation time estimate. The following gives the pseudo-code for the `Arrival.Generator` process.

```
Process Arrival.Generator [SIMSCRIPT routine]
    while(number of transaction arrivals < simulation run length)
    do
    {
        wait exponential time with parameter INTERARRIVAL.TIME.
        generate new transaction.
        assign to transaction :
            arrival time, transaction type, deadline.
        check the Ready Queue.
        if( Ready Queue is empty AND at least 1 MPL token free)
        {
            grab free MPL token.
```

[1]our experiments associated a worst-case computation time estimate associated with each transaction type.

```
            /*make new transaction active immediately*/
            activate Process Active.Transaction now.
         }
        else
            place this new transaction in Ready Queue.
    }loop
end of Arrival.Generator
```

3.2.2 Transaction Manager

All arriving transactions are not immediately accepted for processing by the system. If the system has reached its processing capacity, the arriving transactions have to wait in a queue in an *inactive* state. Once, the system is ready to process a transaction, it removes the transaction from the queue and activates it, creating what we call an *active transaction.* This is the role of the Transaction Manager (TM).

The Transaction Manager module assigns a unique *transaction id* to a new arrival. If the system is already full, i.e., *mpl* transactions are already active, the new transaction is placed in a Ready Queue. This is modeled in our TM module by a set of *mpl tokens* and a token queue. A newly arrived transaction grabs a *mpl token* if one is available, otherwise it waits in the token queue. Once the transaction has grabbed a token, it is in the active state and ready for processing. At this point an `Active.Transaction` process is created. If the deadline of the transaction has already expired (i.e., while waiting in the Ready Queue) the transaction is aborted immediately and the transaction miss counters are incremented by one.

In the `Active.Transaction` process, all incomplete operations at the CPU and DISK resources are dropped. These operations have to be rescheduled along with those of the new transaction. The process then passes information such as transaction type and transaction id. to the CCR module. At this point, the process will wait for the CCR and Scheduler to do their computations. It then receives the selected schedule of operations. These operations are released to the DM module in a sequential manner. The process now suspends itself until it is reactivated again either by the commit or abort event of the same transaction. When a transaction is finished processing (i.e., it commits or misses its deadline), its token is released and made available for incoming transactions. (See `Arrival.Generator` and `Active.Transaction` processes.)

```
Process Active.Transaction [SIMSCRIPT routine]
    Note the activation time of arrived transaction
    If(deadline already expired)
    {
        reject transaction.
        increment transaction miss counter by 1.
        jump to A.
    }
    /* if deadline has not expired... */
    discard all operations waiting in CPU and DISK queues as well
    those currently being processed./*these will be rescheduled*/

    pass new transaction information
        to CCR module (C routine: newtrans).
```

```
    get schedule of operations from
        Scheduler(C routine: scheduler).
    activate processes corresponding to each
        operation in the schedule.
    SUSPEND PROCESS.

    /*Reenter here when process is reactivated */
    REACTIVATE when this transaction is
        either committed or aborted.
    increment NUM.TRANSACTIONS.PROCESSED by one.
 A: relinquish MPL token.
    if (Ready Queue is not empty)
    {
    remove first transaction in queue by activating another
       Active.Transaction.
    }
end  Active Transaction
```

3.2.3 Concurrency Controller (CCR)

The CCR is composed of a group of C routines. Some of the main routines are discussed in this section. The routine `initialize` creates the DES model based on simulation input parameters such as the number of resources in the RTDBS, *mpl* of the system, etc. It creates DFAs for each resource and reads in the masking function, M, for the MSC operation. It also creates templates for *mpl* transactions, the maximum possible number of transactions which can be

active at any given time. `destroy_fsms` destroys the DES model of the CCR at the end of simulation. This essentially involves deallocating memory reserved for the various transaction and resource DFAs and other book-keeping information.

Whenever a new transaction is activated (see process Active.Transaction), the C routine `newtrans` is called. This routine creates a DFA for the newly arrived transaction and shuffles this DFA with the other active transaction DFAs to obtain the plant T; it then shuffles the resource DFAs to obtain R, computes $T \square^M R$ to obtain the specifications, and finally extracts the set of legal traces from the result. See Appendix A for an example run of the CCR. The following is the pseudo-code for `newtrans`.

```
newtrans [C routine]
    create a FSM for the new transaction with information
        obtained from the TM.
    shuffle this FSM with other active transaction FSMs in their
         current state to get T.
    shuffle all resource FSMs to get R.
    compute the MSC of T and R.
    compute legal traces of the resultant FSM.
    pass these legal traces and the deadline information to the
        scheduler (C routine).
end newtrans
```

The CCR also contains several C routines which update the transaction and resource DFAs so that the DES model represents the physical state of the system at any given time. `grab_update` is a routine which fires the grab events in the relevant transaction and resource DFAs whenever a grab process is finished by the

DM module. Similarly, `commit_dead_update` is used to update the DES model whenever a transaction commits or is aborted (due to deadline expiration). This routine destroys the FSM for the particular transaction. It then fires release events in all those resource FSMs which were requested by the transaction while executing. (See the pseudo-code for `grab_update` and `commit_dead_update`).

```
grab_update [C routine]
    get details of grab operation:
        type(read/write),transaction id.(say i)
        and resource id.(say j).

    fire grab event in transaction i DFA.
    fire grab event in resource j DFA.

    add resource j to the transaction i resource list.
end grab_update
```

```
commit_dead_update [C routine]
    get transaction id. (say i).
    delete transaction i DFA.
    delete transaction i deadline information.

    /* release all resources held by transaction i*/
    for (each resource j in transaction i resource list)
         fire release event in resource j DFA.
end commit_dead_update
```

3.2.4 Scheduler

The Scheduler is also composed of a group of C routines. The scheduler module calculates priorities of the various active transactions based on a scheduling policy. We have presented results for an earliest deadline first policy in this chapter. In this policy, the scheduler keeps an array of transaction deadlines which it updates regularly. This array is maintained by the `update_trans_deadlines` routine. Every time a new transaction becomes active, this routine stores the new deadline and updates the deadlines of the already active transactions. Based on this deadline information, another routine, `calculate_priorities` calculates a priority order of transactions. The scheduler now makes two *selection passes* over the list of legal traces it received from the CCR module. In the first pass (this is performed by the routine `pass_one`), it examines the first operation of each trace. If the operation does not belong to the transaction with the highest priority, the trace is discarded. This greatly reduces the number of traces to be examined in the second pass.

The routine `pass_two` examines the second operation of each remaining trace. If it finds a trace whose second operation also belongs to the highest priority transaction, that trace is immediately selected for execution and returned to the Active.Transaction process in the TM module. It is possible that none of the traces have their second operation belonging to the highest priority transaction. In this case, the scheduler chooses the trace, t, whose second operation belongs to a transaction with higher priority than all such transactions. In other words, there is no other trace, r, whose second operation belongs to a transaction which has a higher priority than the one corresponding to the second operation in t[2].

[2]We do not examine the whole trace, in order to reduce computational complexity.

The following is the pseudo-code for the Scheduler.

```
scheduler [C routine]
    get deadline information about new transaction.
    call update_trans_deadlines.
    call calculate_priorities.
    call pass_one.
    call pass_two.
    return selected trace to Active.Transaction.
end scheduler
```

3.2.5 Data Manager

This module takes the list of scheduled operations and processes them. (See pseudo-code for Grab, Commit and Deadline.Expired SIMSCRIPT processes.) It is assumed that each operation consumes some specified amount of CPU time and DISK time. Therefore each operation passes through a CPU resource and DISK resource. The simulator can vary the number of each of these resources, i.e., multiple CPU and DISK systems can be simulated. Note that whenever a Commit or Deadline.Expired process finishes, the corresponding Active.Transaction is reactivated.

```
Process Grab [SIMSCRIPT routine]
    request CPU and wait.
    once CPU is granted, expire time = CPUtime.
    relinquish CPU.
```

```
    request DISK and wait.
    once DISK is granted, expire time = DISKtime.
    relinquish DISK.

    /*update relevant transaction and resource DFAs*/
    call grab_update (C routine).
end Grab

Process Commit [SIMSCRIPT routine]
    request CPU .
    expire CPUtime.
    relinquish CPU.
    request DISK.
    expire DISKtime.
    relinquish DISK.

    cancel Deadline.Expired process for same transaction.
    /*update the DES model of CCR*/
    call commit_dead_update (C routine).
    reactivate the Active.Transaction process.
end Commit

Process Deadline.Expired [SIMSCRIPT routine]
    abort the transaction :
        destroy all operations(grabs and commit).
```

```
    increment the miss counters by one.
    /*update the CCR's DES model*/
    call commit_dead_update (C routine).
    reactivate the Active.Transaction process.
end Deadline.Expired
```

3.2.6 Report Generator

This module prints out the input parameters for each simulation followed by a formatted display of all the output parameters. The output mainly consists of transaction misses, which include misses in the Ready Queue as well as those transactions which could not finish execution inside the system. Other information recorded includes CPU and DISK utilization, average queue lengths of Ready Queue, DISK and CPU queues and average number of active transactions.

RTSIM also produces output which traces the execution of each transaction as it enters and leaves the system. The user can dump this output in a file and trace all the steps in the Synthesis Algorithm (see section 2.2.4) right from the creation of a transaction DFA to the scheduling of operations and their subsequent execution by the DM module.

3.3 Simulation Results

As described in the previous section, the simulator RTSIM collects statistics for various input parameters such as transaction inter-arrival times, slack ratio, number of CPUs, number of DISKS and *mpl* of the TM in the RTDBS. To show how RTSIM works we present the results for a (2pl, earliest-deadline-first)

CCR-scheduler pair simulation run on RTSIM.

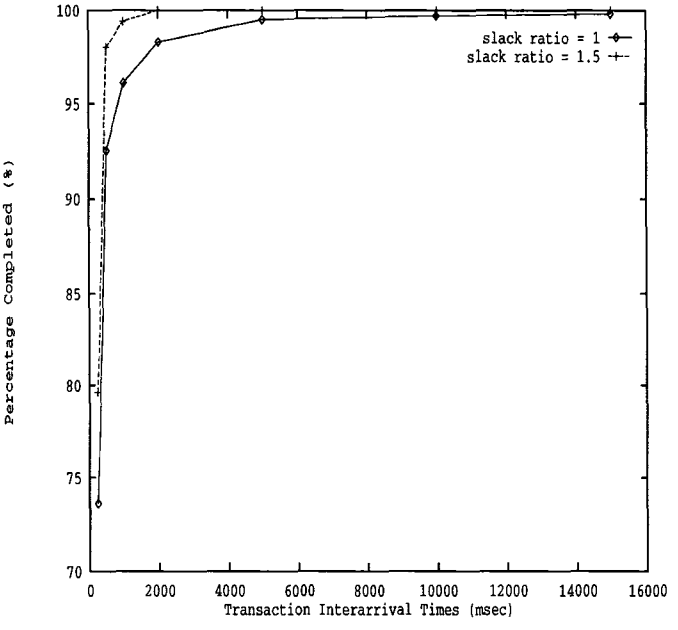


Figure 3.2: Completion Rates for Transactions (1 CPU, 2 Disks)

The transaction and resource DFAs used by the CCR were similar to those described in sections 2.2.2 and 2.2.3. A number of different DFAs were created, each of which represented a different transaction type. For the simulation results presented we used 5 different types of transactions. We ran simulations for two configurations - 1 CPU, 1 DISK; and 1 CPU, 2 DISKs. For each transaction type we had a worst-case computation time estimate. The slack ratio has a very similar meaning to that used in [52] and basically denotes the "difficulty" of schedulability of transactions. It is a multiplication factor used to compute actual deadlines for each transaction type. We varied inter-arrival times and slack ratios to study the impact of those parameters on the system's performance.

The performance was measured using the percentage of transactions which

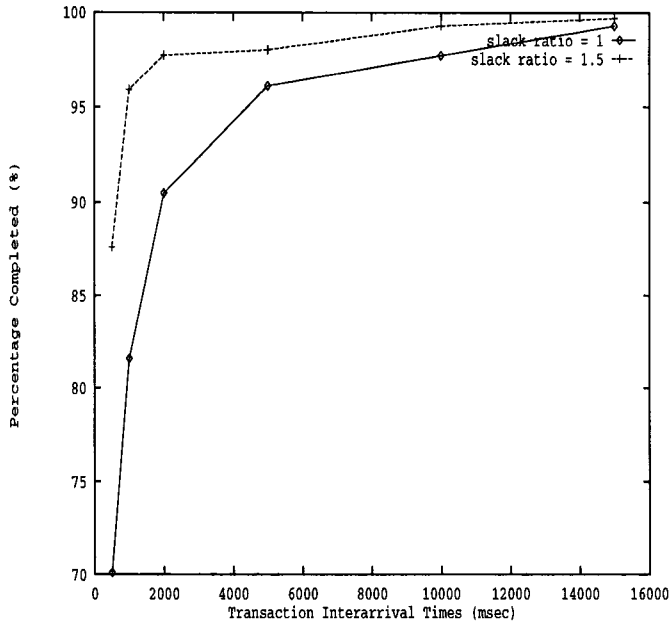


Figure 3.3: Completion Rates for Transactions (1 CPU, 1 Disk)

completed in time. In Figures 3.2 and 3.3, we show the results for (2pl,earliest-deadline-first) CCR, scheduler pair simulation runs on RTSIM for the two different system configurations.

The results were as expected. As transaction inter-arrival times increase, a larger fraction of transactions are completed in time. Also, a larger fraction of transactions are completed (given the same inter-arrival time) for higher slack ratios than lower ones. This is because with higher slack, more sequences of interleaved transaction operations (i.e., traces) satisfy the timing constraints, and therefore, the transactions are "easier" to schedule. To compare the performance of multiple CC-scheduler pairs, one would only have to construct the required CCRs and schedulers in RTSIM and feed identical workloads to these pairs.

Chapter 4

Supervisory Control Theory: A Promising Tool for RTDBS Analysis

In this chapter, we present a control theory of discrete-event systems called supervisory control theory (SCT). SCT was initiated by Ramadge and Wonham [57] in the early 1980's and has been the subject of extensive research since then. SCT been used in diverse application areas including manufacturing, communication networks, database management and transportation systems.

Our goal here is to present convincing arguments to show why supervisory control theory is a promising tool for the analysis and evaluation of transaction execution in RTDBS. We first provide an introduction to SCT. Then we show how the CCR module in our RTDBS can be implemented by reformulating the Synthesis Algorithm as a solution of a SCT *supervisor synthesis* problem. Then we discuss the application of SCT in more complex situations arising in RTDBS

such as the presence of user-initiated transaction aborts and the integration of the CCR and scheduler modules.

4.1 Basics of SCT

The essence of control theory research may be stated through the following problem statement: Given a *plant model* (description of the system to be controlled) and *specifications* (constraints) that need to be satisfied, the goal is to *synthesize* (design) a *supervisor* or controller that observes the plant and applies commands such that the behavior generated by the plant fulfills the given specifications. The branch of control theory that deals with discrete systems (i.e., systems where *distinguished signals* (discrete events) are observed at countable asynchronous times in the time domain) is known as supervisory control theory (SCT) of Discrete Event Systems.

A DES, by itself, is simply a spontaneous generator of event strings with no means of external control. Let us assume that some of these events can be disabled, i.e., prevented from occurring, when desired. This would permit us to influence the evolution of the DES by preventing the occurrence of some specific events at certain points in time. To model such control we partition the event set (also called the *alphabet*), Σ, into *uncontrollable* and *controllable* events, i.e., $\Sigma = \Sigma_u \cup \Sigma_c$. The events in Σ_c can be disabled by the DES controller at any time while those in Σ_u can *never* be disabled. Uncontrollable events correspond to physically inevitable events such as machine breakdown, link failure in a communication network etc.

Consider a DES G which generates events spontaneously from an event set Σ

where $\Sigma = \Sigma_c \cup \Sigma_u$. To control G we could have a controller S track the evolution of G by observing the event string of previously generated events output by G. Based on its information about the state of G, S would generate a set of *disabled* events, say γ. γ serves as the control input to G. All events not present in γ (but which can be executed by G itself in its current state) are considered *enabled* and G is allowed to execute any one of these events. Obviously, none of the uncontrollable events possible at the current state of G would be contained in γ since they cannot be disabled by the controller. The controller S would thus perform a restrictive control function by prohibiting certain events at certain times as G evolves. The control action would therefore be a sequence of γ's generated by the controller in response to the observed event strings generated by G. In this way we have a closed-loop system with the event strings output by G serving as input to S. S, in turn, sends γ as a control input to G based on the observed event string. (See Figure 4.1 .)

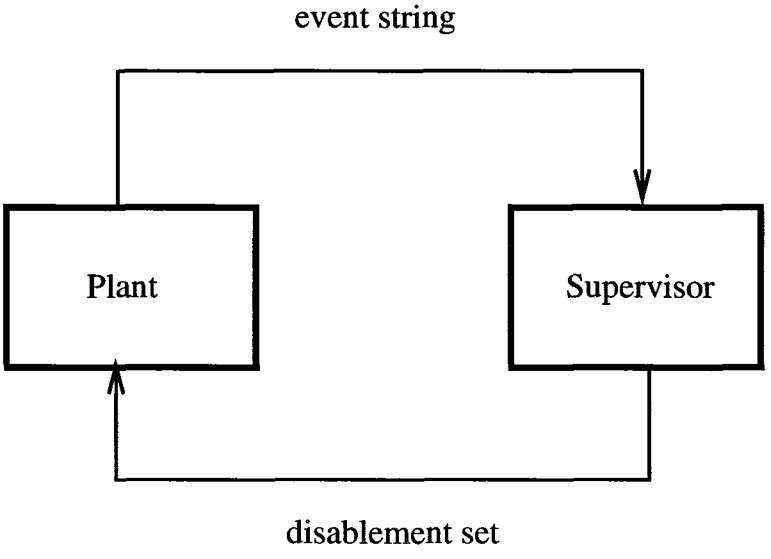


Figure 4.1: Supervisory control of a DES

Here, G, the object to be controlled, is called the plant and S is called the controller or *supervisor* (in SCT terminology). Notice that we clearly distinguish between the controlled and controlling components of the system. This is a standard practice in control theory in general. The supervisor can be implemented either as a mapping $\gamma : \Sigma^* \mapsto 2^{\Sigma}$ (where Σ^* is the set of all strings of events in Σ) as another DES which restricts the plant DES by requiring synchronization on common events (e.g., synchronous composition as explained in section 2.2.1). The latter approach is usually used when DESs are modeled as DFAs.

We now formalize the basic ideas discussed above. Since we modeled DESs as DFAs in Chapter 2 we will continue to discuss SCT in the framework of automata theory. However, the reader should be aware that SCT has been developed for other DES modeling frameworks such as Petri nets [53], temporal logic [14, 44, 54, 33], process algebra and trajectory models.

4.2 Notation and terminology

The plant, the DES to be controlled, is represented as a deterministic finite automaton(DFA). Letting P denote the plant, it is represented as a 5-tuple:

$$P = (Q, \Sigma_P, \delta, q_0, Q_m)$$

where Q is the state space of P (Q is finite, if P is a DFA); Σ_P is the finite *alphabet* or event set; $\delta : \Sigma_P \times Q \mapsto Q$ is the partial state transition function; $q_0 \in Q$ is the initial state of P; and $Q_m \subseteq Q$ is the set of marked or final states which correspond to the completion of some significant activities in the plant P.

The behavior of the plant is described by the strings of events or *language* it

can generate. Formally, the languages *generated* and *recognized* by P, denoted by $L(P)$ and $L_m(P)$ respectively, are defined as :

$$\begin{aligned} L(P) &= \{s \in \Sigma_P^* \mid \delta(s, q_0)!\} \\ L_m(P) &= \{s \in \Sigma_P^* \mid \delta(s, q_0) \in Q_m\} \end{aligned}$$

where Σ_P^* is the set of all strings of events defined over Σ_P, δ has been extended naturally to map a string of events to a resultant state, i.e., $\delta : \Sigma_P^* \times Q \mapsto Q$, and '!' is read as 'is defined'.

The set of prefixes of a language $L \subseteq \Sigma^*$ is given by

$$pref(L) = \{v \in \Sigma^* \mid \exists v' \in \Sigma^*,\ vv' \in L\}$$

where Σ^* denotes the set of all words over Σ and the concatenation of the words v and v' is denoted by vv'. If $pref(L) = L$, the language L is called *prefix closed.* The language $L(P)$ is prefix-closed by its definition.

In the database scenario, we consider the shuffle product of the transaction DFAs as the plant. We model various transaction operations such as acquisition or 'grab' of a resource, release of a resource, commit operation etc. as discrete events. It so happens that, in the RTDBS problems considered above, there are no uncontrollable events because the supervisor (CCR) can reject (i.e., disable) all these operations.

We want to restrict the plant behavior so that certain specifications are met. A specification is also expressed as a language, usually denoted by K. A *supervisor* or controller for P (in our case, the CCR) is defined as the pair $S = (P, \gamma)$, where $\gamma : L(P) \mapsto P(\Sigma_c)$ is a feedback mapping, such that for all $v \in L(P)$, $\gamma(v)$ is the set of events which are disabled after the supervisor has observed v.

The supervisor controls the behavior of P such that $\forall v \in L(P)$,

$$\gamma(v) = \{\sigma \in \Sigma_c \mid v\sigma \in L(P) \wedge v\sigma \notin K\}$$

i.e., $\gamma(v)$ gives the set of disabled events at the state reached by the execution of string v. Note that this set can never contain any uncontrollable event. $L(P, \gamma)$ is the closed-loop language or the language of the plant P under *supervision* by S. (See Figure 4.1.)

One of the basic problems addressed in SCT is the existence of a supervisor given the specification, K. This problem is addressed through the controllability of languages. A language $K \subseteq L$ is *controllable* with respect to L if $pref(K)\Sigma_u \cap L \subseteq pref(K)$.

Based on the above, one of the most fundamental results of SCT may be stated thus:

Existence Theorem: *Let $K \subseteq L$ be a language. There exists a supervisor $S = (P, \gamma)$ such that $L(P, \gamma) = K$ if and only if the language K is nonempty, prefix-closed and controllable w.r.t. P.*
See [58] for proof.

Note that the above result treats S as a *feedback map.* As mentioned earlier, S is often implemented as another DFA which restricts plant behavior to a specified language K through synchronous composition.

Finally, consider the languages L_m (*marked language*), E (*legal language*) and A (*minimally accepted language*), such that:

$$\emptyset \neq A \subseteq E \subseteq L_m \subseteq L$$

Then the supervisory control problem may be phrased as: *Find a prefix closed language K which is controllable with respect to L and satisfies the following two requirements:*

1. $A \subseteq K \cap L_m \subseteq E$

2. $pref(K \cap L_m) = K$

Requirement (1) imposes certain bounds on the marked words contained in the closed loop language. The second requirement, referred to as the *nonblocking* condition says that given any string s in the language, it is always possible to extend s to another string v which is a marked string (i.e., v takes us from the initial state to a marked state).

The are several approaches to solving the above problem. Discussion of these is outside the scope of this section. The reader is referred to [59] for a comprehensive survey.

To summarize, the ideas presented in the above discussion are the following:

- The plant behavior is modeled as a language over an alphabet of event labels;

- Based on the characteristics of the language (e.g., emptiness, prefix closure etc.) certain controllability properties can be proved regarding the specification language, K;

- The *supervisor* controls the plant by disabling controllable events, thereby restricting the behavior of the plant (i.e., the language) such that certain specifications are satisfied. The supervisor is usually implemented as a

DFA which is composed with the plant DFA using *synchronous composition.*

The SCT framework discussed thus far corresponds to the original setup of the supervisory control problem of centralized control under full observation. This means that in the systems modeled using the above approach, the supervisor could observe the entire plant behavior (i.e., all events in the plant alphabet) and could disable all controllable events of the entire plant. Related work can be found in [58, 72, 38]. The basic model was then extended to situations where the supervisor could only observe part of the plant event alphabet [37, 40, 12, 71] and where the supervisor had only partial state information available [56].

To make the control of the increasingly complex man-made (discrete-event) systems more manageable, various architectural methods have been proposed. One approach is that of modular design. [73, 37] discuss modular design under full observation which leads to modular design under partial observation (i.e., decentralized control) as discussed in [37, 39, 41]. Another approach is that of hierarchical control, where [77] explores distributing system complexity over vertical layers as opposed to horizontal layering in decentralized control [61, 40, 13].

Research on the control of real-time systems such as computers, robots, air traffic controllers etc. is another extension of SCT. See the references in section 1.2.2. More recently, a new framework, which can model time-driven open-loop control and non-deterministic controllers, has been developed in [66]. This framework also promotes reusability of various controllers using object-oriented design principles.

4.3 An SCT Formulation of the CCR Synthesis Algorithm

In this section we show that our approach is a special case of the supervisory control approach by demonstrating that our model fits the SCT framework. Consider T, the unrestricted operation of several transactions, as the plant; and R, the composite resource model, as the *specification* (see section 2.2.2 and section 2.2.3). The specification, K, may be phrased as follows: *"the executing transaction cannot lock resources in a mode which conflicts with the lock mode of another transaction grabbing the same resource"*. Examples of conflict include shared-exclusive or exclusive-exclusive lock conflict. As mentioned in the previous section, it is often convenient to implement the supervisor as another DFA (just like the plant) and have the supervisor restrict the plant behavior within the desired limits through synchronous composition (SC) [59]. One may now proceed to obtain the specifications in exactly the same as way we did in the Synthesis Algorithm presented earlier. The MSC of T and R provides us the DFA representation of the specification language, K. Now, since all the events in the DES model of T are controllable (i.e., they can be disabled by the supervisor), the language K is *trivially* controllable. Hence, $T \square^M R$ itself yields the supervisor and also gives the set of legal traces as its marked language. Thus, we have that our approach of modeling transactions and resources as DFAs and synchronizing them can be viewed as a simple application of SCT.

4.4 Possible Application of SCT in More Complex RTDBS Analysis

Our DES model of the RTDBS consisted of only *controllable* events and hence made the SCT formulation of the *supervisor synthesis problem* trivial. We did not have to use a supervisor synthesis algorithm to construct a supervisor. Instead, the DFA representation of the specification K itself served as the supervisor S.

However, there exist more complex situations in an RTDBS which would require the DES model to have *uncontrollable* events in its event alphabet. For example, if we model user-initiated transaction aborts in our DES model, the events corresponding to such aborts would be considered uncontrollable. This is because the *supervisor* would not be able to disable such aborts. In such formulations, one way to arrive at a correct implementation of the CCR module using the synchronization techniques discussed earlier would be to perform a trial-and-error procedure and verify the correctness of the synchronized (composite of T and R) DFA for various models of T and R. It would be much better, however, to have a methodology which establishes the existence of the solution and provides a systematic procedure for arriving at the solution.

In such cases, SCT can contribute substantively. Although it is no longer possible to use K's DFA representation itself as the supervisor, several *supervisor synthesis* algorithms have been proposed in the literature to generate solutions for these non-trivial SCT problems. Using these algorithms, it is possible not only to establish the existence of a supervisor for the DES control problem but also to synthesize supervisors which are correct by construction. The closed loop behavior obtained through the coupling of these supervisors with the plant,

however, is not the same as K, the specification language. Instead, it is the largest controllable subset of K or the *supremal controllable sublanguage* of K. See [32] for an example of transaction models with uncontrollable events.

Another possible application of SCT techniques is using the timed-discrete event systems SCT framework [5] to integrate the timing information (i.e., deadlines) associated with the transactions in the DES model. In our approach, we separated the CC and scheduling functions and therefore our DES model of the RTDBS contained only non-temporal information. However, one could argue that better performance could be achieved by integrating the two modules and solving the composite as a solution of a timed-discrete event (TDES) system control problem. In [5], a TDES models the passage of time through *ticks* of a clock where the *tick* event is treated as a separate event label in the automaton representation of the DES model of the transactions. The existence conditions and construction techniques of supervisors for TDES is shown in [5]. Also, the use of a controlled timed-petri net framework in the analysis of transaction scheduling in RTDBS is demonstrated in [63]. Thus, SCT seems to be a promising tool for the analysis of RTDBS in the future.

Chapter 5

Conclusion

The focus of this work has been to describe a novel approach to model and analyze concurrency control and scheduling policies in RTDBSs. The approach is based on the theory of discrete event dynamical systems (DEDSs), automata theory and formal languages. This approach has been successfully used in several other DES control domains such as manufacturing plants and communication protocol verification.

We have modeled transaction execution in a RTDBS using DFAs and extracted the effect of concurrency by composing these DFAs through SC and MSC. From this composition we extract legal traces and forward them to the scheduler. The scheduler chooses one particular trace that best fits the priority order as determined by the scheduling policy used. We also show how the above approach can be viewed as a SCT problem and solved (albeit trivially). Towards this end, we also show the potential of the application of SCT in various RTDBS settings.

Aside from the modeling of an RTDBS as a DEDS, this work makes the case

for separation of the CC and scheduling modules. This allows the evaluation of multiple CC-scheduling combinations without having to change the basic simulation model. Also, the DES formulation provides a uniform framework to the hitherto empirical studies of performance evaluation.

An additional goal of this research is to motivate the use of the several new DES models which have been proposed in the recent literature such as the timed-DES models [5] and DES model interconnection using masked composition [66] in RTDBS analysis.

Appendix A

Example Run of Synthesis Algorithm

We illustrate the synthesis algorithm with the aid of a "toy" example.

Consider a transaction T_1 in a database with two resources (say data items) R_1 and R_2. The DFAs corresponding to T_1, R_1 and R_2 are shown in figures A.1A, A.1B and A.1C respectively. In particular, these figures reflect the state of the DFAs just following the execution of the event $g_1[1]$ by T_1. These states are labelled "current state" in figure A.1.

Assume at this point a new transaction T_2 enters the system. Assume T_2 just wants to write the value (i.e., exclusive grab) of R_2. The steps of the synthesis algorithm, as given in section 2.2.5 are as follows:

1. Create DFA for T_2: The DFA of T_2 is shown in figure A.2.

2. Compute the composition for the transactions: In this step we compute the shuffle of T_1 and T_2, taking into consideration that T_1 is in its current

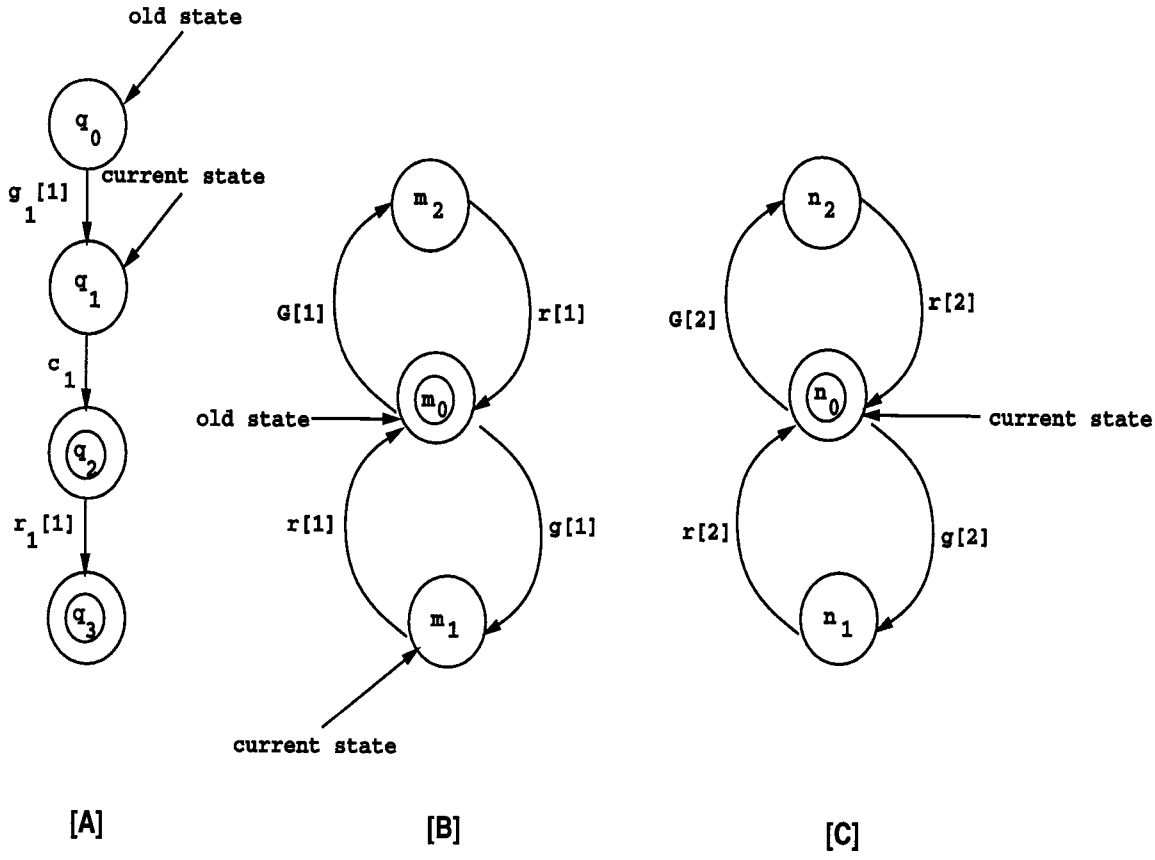


Figure A.1: DFAs for T_1, R_1 and R_2

state. This composition $T = T_1||T_2$. T is shown in figure A.3A.

3. Compute the composition of the resources: Here we compute the composition $R = R_1||R_2$. Note that R is computed taking into consideration that the two resources are in their current states. R is shown in figure A.3B.

4. Compute MSC of T and R: We compute $C = T\square^M R$, where M is defined as:

- $M(r_1[1]) = r[1]$
- $M(G_2[2]) = G[2]$

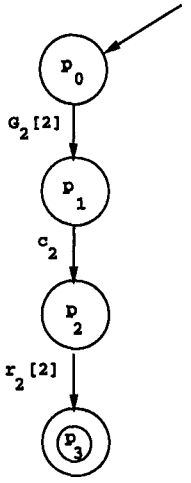


Figure A.2: DFA for T_2

- $M(r_2[2]) = r[2]$

C is shown in figure A.3C.

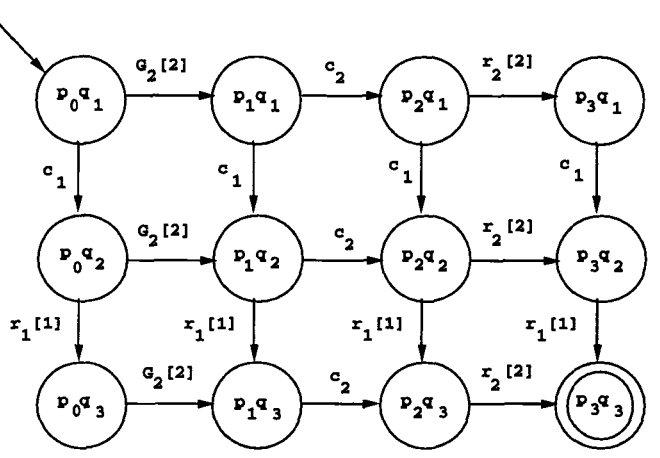
5. Extract legal traces from C: Legal traces correspond to those which follow the strict 2-phase locking protocol (as imposed by our resource and transaction models). The following legal traces are obtained from C:

(a.) $c_1, G_2[2], c_2$;

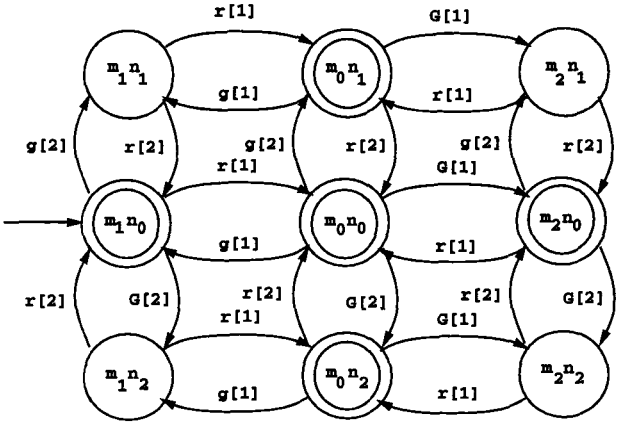
(b.) $G_2[2], c_1, c_2$; and

(c.) $G_2[2], c_2, c_1$.

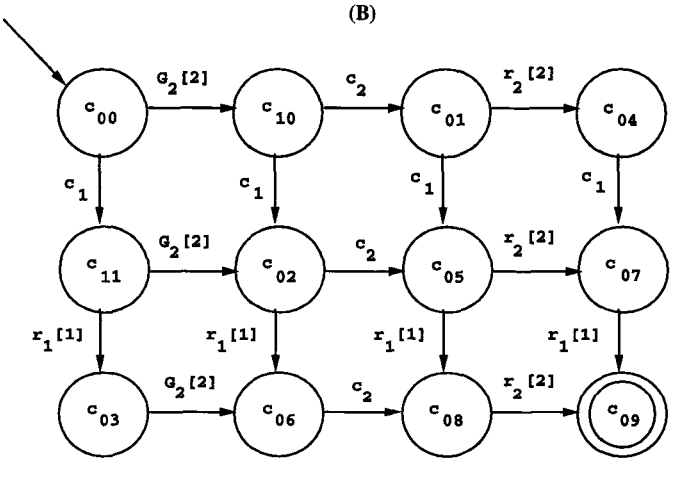
Note that the release (i.e., the $r_i[j]$) operations have been projected out of the traces as they are not explicitly scheduled by the scheduler. The above trace list is now forwarded to the scheduler which then chooses one based on the priority order it computes from the deadlines of T_1 and T_2.



(A)



(B)



(C)

Figure A.3: Transaction and Resource Compositions

Appendix B

Scheduler Example

In this section we will take the list of traces computed by the CCR, as shown in Appendix A, and select one for scheduling based on the *earliest-deadline-first* scheduling policy. Let us assume that the transaction T_1, described in the previous section, had a deadline $d_1 = 100ms$ when it entered the RTDBS. The scheduler immediately stores d_1 in its deadline information array for active transactions. Since there was only one transaction, a serial execution of the operations of T_1 was ordered by the scheduler. Let us assume that 50ms expired by the time the first operation, $g_1[1]$, was finished by the DM. At this point, transaction T_2, also described in the previous section, entered the RTDBS with a deadline $d_2 = 100ms$. The scheduler immediately noted down d_2. The CCR then executed the synthesis algorithm again and extracted the three traces described as (a.),(b.) and (c.) in Appendix A. The scheduler now calculates priorities for the two transactions. As 50ms have expired since T_1 began, d_1 is updated to $100 - 50 = 50ms$. Since $d_1 < d_2 = 100ms$, T_1 is assigned a priority of 1 and T_2, a priority value 2. Based on this priority order, the scheduler examines traces

(a.), (b.) and (c.). Since (a.) fits the priority order best (because it executes a T_1 operation first), the scheduler picks this trace to schedule operations.

Bibliography

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions with disk resident data. In *Proceedings of the 15th VLDB*, 1989.

[2] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: Performance Evaluation. *ACM Transactions on Database Systems*, 1992.

[3] Silvano Balemi. *Control of Discrete Event Systems: Theory and Application.* PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1992.

[4] P. Bernstein, V. Hadzilakos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[5] B.A. Brandin and W.M. Wonham. The supervisory control of timed discrete event systems. In *Proceedings of the IEEE Conference on Decision and Control*, Tucson, Arizona, December 1992.

[6] B.A. Brandin, W.M. Wonham, and B.Benhabib. Manufacturing cell supervisory control - a timed discrete-event system approach. In *International Conference on Robotics & Automation*, pages 531–536, Nice, France, May 1992.

[7] Bertil A. Brandin. *Real-Time Supervisory Control of Automated Manufacturing Systems.* PhD thesis, University of Toronto, Toronto, Canada, February 1993.

[8] Y. Brave and M. Heymann. Formulation and control of real time discrete event processes. In *Proc. of the 27th Conf. on Decision and Control*, pages 1131–1132, Austin, Texas, December 1988.

[9] S.D. Brooks, C.A.R. Hoare, and A.W. Roscoe. *A Theory of Communicating Sequential Processes*, volume 197, pages 281–305. Springer Verlag, 1985.

[10] S.D. Brooks and A.W. Roscoe. *An Improved Failures Model for Communicating Processes*, volume 31, pages 560–599. Springer Verlag, 1984.

[11] M.J. Carey, R. Jauhari, and M. Livny. Priority in dbms resource scheduling. In *Proceedings of the 15th VLDB*, 1989.

[12] H. Cho and S. I. Marcus. Supremal and maximal sublanguages arising in supervisor synthesis problems with partial observations. *Mathematical Systems Theory*, 22:177–211, 1989.

[13] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33:249–260, March 1988.

[14] E.A. Emerson and J. Srinivasan. *Branching Time Temporal Logic*, volume 354, pages 123–172. Springer Verlag, 1988.

[15] K. Eswaran et al. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11), 1976.

[16] Stanley B. Gershwin. Hierarchical Flow Control: A Framework for Scheduling and Planning Discrete Events in Manufacturing Systems. In *Proceedings of IEEE*, volume 77, pages 195–209, January 1989.

[17] J. Haritsa, M. Livny, and M. Carey. Earliest Deadline Scheduling for Real-Time Database Systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1991.

[18] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. Dynamic Real-Time Optimistic Concurrency Control. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1990.

[19] Jayant R. Haritsa, Michael J. Carey, and Miron Livny. On being optimistic about real-time constraints. *ACM PODS*, 1990.

[20] M. Hennesey. *Algebraic Theory of Processes.* MIT Press, 1988.

[21] C.A.R. Hoare. *Communicating Sequential Processes.* Englewood Cliffs, 1985.

[22] G. Hoffman, C. Shaper, and G. Franklin. Discrete event controller for a rapid thermal multiprocessor. In *Proc. of the American Control Conference*, volume 3, pages 2936–2938, Boston, MA, June 1991.

[23] G. Hoffmann and H. Wong-Toi. The Input-Output Control of Real-Time Discrete Event Systems. In *Proc. of the IEEE Real-time Symp.*, pages 256–265, Phoenix, AZ, December 1992.

[24] Debra J. Hoitomt, Peter B. Luh, Eric Max, and Krishna R. Pattipati. Scheduling Jobs with Simple Precedence Constraints on Parallel Machines. In *IEEE Control Systems Magazine*, pages 34–40, February 1990.

[25] L.E. Holloway and B.H. Krogh. Synthesis of feedback control logic for a class of controlled Petri nets. *IEEE Transactions on Automatic Control*, 35(5):514–523, May 1990.

[26] J.E. Hopcroft and J.D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

[27] J. Huang, J. Stankovic, D. Towsley, and K. Ramamrithnam. Experimental Evaluation of Realtime transaction processing. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1989.

[28] Jiandong Hyang, John A. Stankovic, Krithi Ramamritham, and Don Towsley. Experimental evaluation of real-time optimistic concurrency control schemes. In *Proceedings of the 17th VLDB*, 1991.

[29] Jiandong Hyang, John A. Stankovic, Krithi Ramamritham, and Don Towsley. On using priority inheritance in real-time databases. In *Proceedings of the Real-Time Systems Symposium*, 1991.

[30] W. Kim and J. Srivastava. Enhancing real time DBMS performance with multiversion data and priority based disk scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 222–231, 1991.

[31] M.H. Klein, J.P. Lehoczky, and R. Rajkumar. Rate-Monotonic Analysis for Real-Time Computing. *IEEE Computer*, 27(1):24–32, January 1994.

[32] P. Kozak and W.M. Wonham. Synthesis of Database Management Protocols. Technical Report 9311, Systems Control Group, Department of Electrical Engineering, University of Toronto, 1993.

[33] F. Kroger. *Temporal Logic of Programs.* Englewood Cliffs, 1987.

[34] S. Lafortune. Modeling and Analysis of Transactions Execution in Database Systems. *IEEE Transactions on Automatic Control*, 33:439–447, 1988.

[35] A.M. Law and C.S. Larmey. *An Introduction to Simulation Using Simscript II.5.* CACI Products Company, 1984.

[36] J. Lee and S.H. Son. Using Dynamic Adjustment of Serialization Order for Real-Time Database Systems. In *Proc. IEEE Real-Time Systems Symposium*, December 1993.

[37] F. Lin. *On Controllability and Observability of Discrete Event Systems.* PhD thesis, Department of Electrical Engineering, University of Toronto, 1987.

[38] F. Lin, A. F. Vaz, and W. M. Wonham. Supervisor specification and synthesis for discrete event systems. *International Journal of Control*, 48:321–332, 1988.

[39] F. Lin and W. M. Wonham. Decentralized supervisory control of discrete-event systems. In *Information Sciences*, volume 44, pages 199–224, 1988.

[40] F. Lin and W. M. Wonham. On observability of discrete-event systems. In *Information Sciences*, volume 44, pages 173–198, 1988.

[41] F. Lin and W.M. Wonham. Decentralized control and coordination of discrete-event systems with partial observation. *IEEE Transactions on Automatic Control*, 35(12):1330–1337, December 1990.

[42] Yi Lin and Sang H. Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1990.

[43] C. Liu and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM*, January 1973.

[44] Z. Manna and A. Pnueli. *The Anchored version of the Temporal Framework*, volume 354, pages 201–284. Springer Verlag, 1989.

[45] R. Milner. *A Calculus for Communicating Systems*, volume 92. Springer Verlag, 1980.

[46] J.S. Ostroff. *Temporal Logic for Real-Time Systems.* Advanced Software Development Series. Research Studies Press, Somerset, England, 1989.

[47] J.S. Ostroff and W.M. Wonham. A temporal logic approach to real time control. In *Proc. of the 24th Conf. on Decision and Control*, pages 656–657, Ft. Lauderdale, Florida, December 1985.

[48] J.S. Ostroff and W.M. Wonham. State machines, temporal logic and control: A framework for dicrete event systems. In *Proc. of the 26th Conf. on Decision and Control*, pages 681–686, Los Angeles, California, December 1987.

[49] J.S. Ostroff and W.M. Wonham. A Framework for Real-Time Discrete Event Control. *IEEE Transactions on Automatic Control*, 35:386–397, 1990.

[50] S.D. O'Young. On the synthesis of supervisors for timed-discrete event processes. Technical report, Department of Electrical Engineering, University of Toronto, 1991.

[51] S.D. O'Young. Synthesis of optimal supervisors for telecommunication systems with timing constraints. Technical report, Department of Electrical Engineering, University of Toronto, 1991.

[52] Hweeehwa Pang, Miron Livny, and Michael J. Carey. Transaction scheduling in multiclass real-time database systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1992.

[53] J.L. Peterson. *Petri Nets and Modeling of Systems.* Englewood Cliffs, New Jersey, 1981.

[54] A. Pnueli. *Application of Temporal Logic to the Specification and Verification of Reactive Systems: A Survey of Current Trends*, volume 224, pages 510–584. Springer Verlag, 1986.

[55] B. Purimetla, R.M. Sivasankaran, J.A. Stankovic, K. Ramamritham, and D. Towsley. Priority Assignment in Real-Time Active Databases. Technical report, Computer Sciences Department, University of Massachusetts, 1994.

[56] P. J. Ramadge. Observability of discrete-event systems. In *Proc. 25th Conf. on Decision and Control*, pages 1108–1112, 1986.

[57] P.J. Ramadge and W.M. Wonham. Supervision of discrete event processes. In *Proc. 21st IEEE Conf. on Decision and Control*, volume 3, pages 1228–1229, December 1982.

[58] P.J. Ramadge and W.M. Wonham. Supervisory control of a class of Discrete event processes. *SIAM Journal on Control and Optimization*, 25:206–230, January 1987.

[59] P.J. Ramadge and W.M. Wonham. The control of discrete event systems. In *Proc. IEEE*, volume 77, pages 81–98, 1989.

[60] K. Ramamritham. Real-Time Databases. *International Journal of Distributed and Parallel Databases*, 1993.

[61] Karen G. Rudie. *Decentralized Control of Discrete-Event Systems.* PhD thesis, Department of Electrical Engineering, University of Toronto, 1992.

[62] H. Sanghavi. REPORT: A Software library for Discrete Event Systems and Other Finite State Machine-based Applications. University of Texas, Austin, August 1991.

[63] S.K. Sathaye and B. Krogh. Application of Supervisor Synthesis for Controlled Time Petri Nets to Real-Time Database Systems. In *Proc. Automatic Control Conference*, July 1994.

[64] Lui Sha, Ragunathan Rajkumar, and J.P. Lehoczky. Concurrency control for distributed real-time databases. *ACM SIGMOD RECORD*, 1988.

[65] Lui Sha, Ragunathan Rajkumar, and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 1990.

[66] M.A. Shayman and R. Kumar. A new approach to supervisory control. In *Proceedings of the Allerton Conference on Communication, Control, and Computing*, Monticello, IL, September 1994. to appear.

[67] C. Shen, K. Ramamritham, and J. Stankovic. Resource reclaiming in real-time. In *Proc. Real-Time System Symp.*, 1990.

[68] J.A. Stankovic. Misconceptions About Real-Time Computing. *IEEE Computer*, pages 10–19, September 1988.

[69] John A. Stankovic and Wei Zhao. On Real-time transactions. *ACM SIGMOD RECORD*, 1988.

[70] Y.C. Tay, Nathan Goodman, and Rajan Suri. Locking performance in centralized databases. *ACM Transactions on Database Systems*, 10(4), 1985.

[71] J. N. Tsitsiklis. On the control of discrete-event dynamical systems. *Mathematics of Control, Signals and Systems*, 2:95–107, 1989.

[72] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal of Control and Optimization*, 25:637–659, 1987.

[73] W. M. Wonham and P. J. Ramadge. Modular supervisory control of discrete-event systems. *Mathematics of Control, Signals and Systems*, 1:13–30, 1988.

[74] P.S. Yu, K-L. Wu, K-J. Lin, and S.H. Son. On Real-Time Databases: Concurrency Control and Scheduling. *Proceedings of the IEEE*, 82(1):140–157, 1994.

[75] W. Zhao and K. Ramamritham. Simple and Integrated Heuristics Algorithms for Scheduling Tasks with Time and Resource Constratints. *Journal of Systems and Software*, 7:195–205, 1987.

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

[77] H. Zhong and W. M. Wonham. On the consistency of hierarchical supervisors in discrete-event systems. *IEEE Transactions on Automatic Control*, 35:1125–1134, October 1990.