1998

A real-time active database for high transaction loads and moderate deadlines

Donald Wayne Carr
Iowa State University

INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600

A real-time active database for high transaction loads and moderate deadlines

by

Donald Wayne Carr

A dissertation submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Major Professors: Dr. Terry A. Smay and Dr. Leslie L. Miller

Iowa State University

Ames, Iowa

1998

UMI Number: 9962861

UMI Microform 9962861

Bell & Howell Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

Graduate College
Iowa State University

This is to certify that the Doctoral dissertation of

Donald Wayne Carr

has met the dissertation requirements of Iowa State University

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

Co-major Professor

Signature was redacted for privacy.

For the Major Program

Signature was redacted for privacy.

For the Graduate College

TABLE OF CONTENTS

ACKNOWLEDGMENTS

I thank all the people on my committee, Dr. Leslie L. Miller, Dr. Terry A. Smay, Dr. James A. Davis, Dr. Doug W. Jacobson, and Dr. Johnny S. K. Wong. They have greatly contributed to my professional development, both in the classroom, and with ideas, support and guidance for this research

Special thanks are due to my co-chairs, Dr. Miller and Dr. Smay. Dr. Miller has given unstintingly of his time with ideas and with encouragement, always demanding better work and more detailed descriptions. Without his persistent prodding and encouragement I could never have done this work. Dr. Smay, who has taken valuable time from his retirement, has continually helped and encouraged me throughout the work.

Family and friends have given me great support as well as encouragement and help with ideas about my work. The support and encouragement of my brother, Dan, has been very important. My mother and father have believed in me and encouraged me, and their love has helped me from the beginning. Many other friends have been of great help. I only wish there were space to mention them all.

Most of all, I thank my loving family, my wife, Ana, and son, Ricky. Ana has given so much love and support that it is impossible to thank her enough. So many times, she has given unselfishly of herself, doing housework and caring for our son so that I might work on research, even though she is pursuing a degree herself. My son, Ricky, has been a constant inspiration for me as I have watched him grow and change.

CHAPTER 1. INTRODUCTION

Over the past 10 years, we have seen increasing use of real-time systems to monitor and control large and arbitrarily complex real-world systems. The biggest problem is the fundamental limitation of human capacity for dealing with such complexity. This has led to a considerable amount of research on how to manage complexity for such large systems. There is a growing consensus that the best way for humans to manage such arbitrarily complex systems is through the use of object-oriented design and programming tools [20, 39, 81]. At the same time, there are few object-oriented tools available for the design and development of large real-time systems.

There has been a great deal of research done on active database design. Most of this research has been for relational databases, but there is an increasing amount of work being done on active object-oriented databases. These databases are mostly disk-resident and of little use for real-time systems, but some of the design features and concepts are useful for real-time applications. For real-time databases most of the research is also for relational database systems, but recently there has been some work on object-oriented real-time database systems. Much of the real-time database research (both relational and object-oriented) has focused on very fast, very small, real-time database systems with requirements of less than one millisecond transaction time deadlines. This work ignores a large class of problems: large database systems with

very high transaction loads (on the order of thousands per second) and only moderate deadlines (on the order of tens or hundreds of milliseconds). In the area of control, the very fast, very small databases would typically be used for machine controllers mounted very close to the real-world device or entity being monitored and controlled. A very large database with a high transaction load and moderate deadlines would typically be used to monitor and control a very large real-world system. Examples with such requirements are: an interstate gas pipeline network, a water distribution system, an entire factory, etc. Throughout the text we refer to these applications as *target applications*. We have developed a model for large, real-time active object-oriented databases (REACT) and have implemented the REACT database kernel as a proof of concept. The target applications imply that the system will be memory-resident, but that is not required by the model. We have developed a concurrency control algorithm for REACT and simulated the REACT database model to verify performance. We also present our research on other real-time database issues and REACT features such as: distributed database, confluence, termination, observable determinism, concurrency, missed deadlines, timers, alarming, and history. We also show how REACT can be used to implement real-world control systems.

1.1 Contribution

The current work on real-time databases has largely ignored environments with very high transaction loads and moderate deadlines. To bridge this gap, we have developed a model for large, real-time active object-oriented databases (REACT) and implemented the REACT database kernel as proof of concept. The model is based on encapsulated events and rule objects. Encapsulated events are named events that have a one-to-one correspondence to real-world events instead of being based on database events. Rule objects respond to encapsulated events and replace the traditional rule base. Rule objects are much more efficient than traditional rules because, with encapsulated events, we avoid searching the rule base after performing a database operation. With REACT the task of creating an application is greatly simplified because of the encapsulated event / rule object interface. The tasks of writing the encapsulated event, rule objects and application dependent code are separated. For example, in a control environment, the REACT rule and event system provides a clean separation of duties between the control engineer, the computer engineer, and the electrical engineer.

We have developed a new algorithm for concurrency control for the REACT object database model to take advantage of multi-processor systems and have simulated the REACT object database model for a multiprocessor system using the REACT concurrency control algorithm. We have also developed an algorithm, based

on the REACT object database model, to distribute objects across multiple computers when a single computer cannot provide the required performance. We give a recovery algorithm for the REACT object database model designed to meet the needs of a subset of the target applications. We have also developed and implemented a set of techniques (tools) to allow a user to test a target database to determine whether or not it satisfies properties of termination, confluence, and observable determinism. We also look at features needed for typical real-world applications for which REACT was designed. We show how REACT can be used to implement large real-world control systems.

CHAPTER 2. RELATED RESEARCH

Lortz et al. [68] has looked at the use of an object-oriented database for real-time control. The system they developed was called MDARTS (Multiprocessor Database Architecture for Real-Time Systems). This is a framework for developing object-oriented data management services on multiprocessor computing platforms and is intended for hard real-time systems. As with REACT, this database does not support ad hoc queries, but supports fast transaction times for typical transactions required by typical target real-time applications. While REACT is for large real-time database systems with moderate deadlines, MDARTS is for very small real-time database systems with deadlines under one millisecond. They claim a guarantee of transaction times less than 100 microseconds which would only be possible for very small databases with light loading. Unlike REACT, MDARTS is not active. This work also ignores the issue of recovery which is very important for the class of real-time applications for which REACT was designed.

Another important area of related work was that of active object-oriented databases not designed for real-time applications [5, 29, 30, 32, 41, 42, 54, 59]. R. Agrawal et al. at AT&T have done a considerable amount of work based on ODE (Object Database Environment) [32, 41, 42, 54, 59]. This is a disk-based system that falls into the category of a persistent object add-on for an object-oriented programming language (C++). The rules in ODE are the same as rules for relational

systems except that the events are based on operations on objects instead of relational operations. The basic events are before member function execution, after member function execution, before insert, after insert, before delete, after delete, etc. They also support a syntax for composite event detection. Unlike REACT, the member functions are executed by the client after objects are retrieved from the server. This database is not meant for real-time applications. S. Chadravarthy et al. at the University of Florida have done work on Sentinel [13, 29, 30], an active object-oriented disk-resident database. Their work is very similar to ODE except that they use event objects to detect events and rules are considered objects. The event objects, however, still use the same low level database events as ODE and the rules are still declared in the same manner. This is also a persistent object add-on for C++. As with ODE, this database is not suitable for real-time applications.

Aiken et al. have done research on techniques for predicting the behavior of database rules for a relational database with a traditional rule base [8,9]. They develop conservative algorithms to determine if the properties, termination, confluence, and observable determinism, hold for a given rule base. The algorithms they develop are based on execution graphs derived from analyzing the rule base. We have also developed conservative algorithms for determining if the properties, termination, confluence, and observable determinism hold, but for an object oriented database that uses encapsulated events and rule objects to implement active functionality.

Ulusoy et al. have done research on speeding up concurrency control by

preprocessing transactions that run on a memory-resident real-time relational database [98]. They require all possible transactions to be processed off line to pre-determine the read set and write set before they can be submitted for execution. We do similar pre-processing to speed up concurrency control. However, because the REACT model is object-oriented and fully supports encapsulation, all possible database operations are predefined as part of the database. For this reason, we can pre-process the database rather than individual transactions to speed up concurrency control algorithms. This has a similar effect on speed up and also eliminates the need to pre-define and pre-process all possible transactions. Our algorithm is based on execution groups as described in Chapter 5.

Since REACT is a memory-resident database we also looked at general research for memory resident databases [36, 40, 44, 56, 66, 98]. A key issue with any memory-resident database is a recovery algorithm to restore the database to a valid state in the event of system crash. The existing work ignores one of the major requirements in many real-time systems: the need to be up and running as soon as possible (within seconds) after a system crash or hardware failure. Existing systems are only designed for high throughput and the ability to completely recover all transactions in the event of failure. As an example, Jagadish et al. have done research on recovering from crashes with memory-resident databases [56]. They write a "redo log" to disk, as part of every transaction, in able to insure that every transaction can be recovered in case of a system crash. This does guarantee that each and every transaction that

occurred before the crash can be restored but at the cost of reducing the performance such that it would not be adequate to support the loads required by the applications for which REACT was designed. Also, recovery time is much greater since all log file entries must be applied to the database before transaction processing can resume. With control systems, it is more important to be up and running as soon as possible than to completely recover all database transactions.

Another area of related research is for real-time transaction processing [1, 2, 11, 24, 46, 49, 52, 79, 84, 86, 91, 93, 96, 98, 100]. Almost all of this research was for transaction loads much less than thousands of transactions per second. None of this research pre-processes the database to speed up concurrency control as is done with REACT. Also, almost all of this research assumes that the time to perform concurrency control is insignificant in comparison to the time to execute transactions. With the high transaction loads required by the applications for which REACT was designed, the time to execute concurrency control algorithms becomes very significant. As an example, one of the most quoted papers on real-time transaction processing by Garcia-Molina et al. [1] does not take into account the time to execute the lock manager, conflict manager, and deadlock detection manager. This is possible since the transaction loads supported are less than 10 transactions per second. With this rate of transaction processing, the time to perform concurrency control algorithms is insignificant. In contrast, since they assume the time to execute concurrency control algorithms is insignificant, they focus their research on scheduling algorithms.

CHAPTER 3. THE REACT OBJECT DATABASE MODEL

We have developed an object-oriented database model for active real-time systems that would be not only easy to use but would also have a one to one correspondence with events in the real-world instead of with events being based on database operations. To achieve this, we developed the notion of encapsulated events which are named events that are detected internally by objects with private algorithms for detecting events. As data flows into the database from the real-world we generate example events such as "value changed" or "fault detected" which correspond to real-world events instead of database events such as "after update" or "after insert". To replace traditional rules we also developed rule objects that are fired by encapsulated events and not database operations. The REACT database model is based on the standard object model together with encapsulated events and rule objects to add active capability to the database. The following sections provide a more formal discussion of this model.

3.1 REACT Database Model and Definitions

The user view of the database is a set of object classes. For each object class there will be one member function for each update/query operation that is valid for

that object class. We define two kinds of member functions. An *update member function* (Figure 3.1) can cause changes to the database. A *query member function* (Figure 3.2) can never cause any changes to the database. (Note the difference in the graphical symbolism for the two function types.)

Since the REACT model is for real-time database systems, we separate object attributes into two categories: those that change in real-time (real-time attributes) and those that do not (configuration attributes). An attribute is a *real-time attribute* if it can change in real-time in response to updates or events. An attribute is a *configuration attribute* if it can never change in real-time and is only set when the object is created or configured. This turns out to be important for the efficient implementation of recovery algorithms as discussed in Chapter 5.

Because REACT is active, database operations can trigger events that cause actions. With REACT we use a publish and subscribe model based on encapsulated

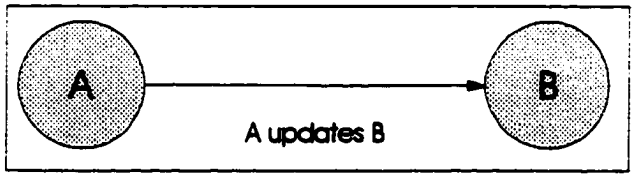

Figure 3.1 Notation for update member function

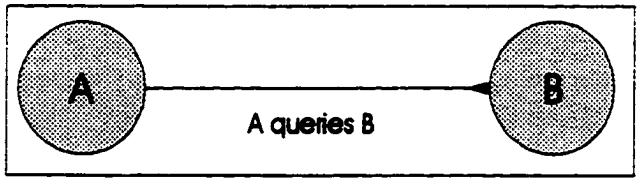

Figure 3.2 Notation for query member function

events. We will say that an object *publishes* an event, if that object detects the event and notifies the DBMS when the event occurs. We will say that an object *subscribes to* an event if that object requests to be notified by the DBMS when the event occurs. Encapsulated events are the foundation for the active features of REACT. The notion is formally defined as follows:

Definition 3.1. Assume an object O with event E. E is an *encapsulated event* (Figure 3.3) if O publishes E and keeps the detection algorithm private, revealing only a name, signature, and a description.

With the REACT database model, all event handling and notification are handled by the DBMS kernel, so objects are only responsible for notifying the DBMS of the occurrence of an event. An object *triggers* an encapsulated event when that object detects the encapsulated event and notifies the DBMS of this occurrence. After an object triggers an event, the DBMS delivers an event notification to all subscribers.

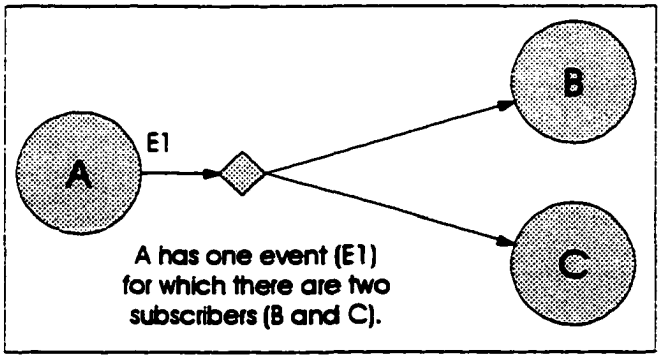

Figure 3.3 Notation for encapsulated events

The *event handler* for an event is the block of code that is executed by a subscriber on delivery of event notification. A rule is *fired* when an event notification is delivered and the event handler is executed. We will now define rule objects which replace the notion of a rule base in a traditional active database. As will be examined in Chapter 7, the use of encapsulated events and rule objects provides a speedup that is critical to the success of a real-time active database.

Definition 3.2. Assume an object O. O is a *rule object* if O subscribes to encapsulated events and executes event handlers that perform actions (execute update member functions and/or trigger events) based on conditions (from executing query member functions).

Because REACT is a real-time database, some objects will have the ability to interact with the real-world through device drivers. We will say that an object is an *interface object* if this object is directly tied to some entity in the real-world, through a device driver, either for input or for output. An *observable action* is any action which can cause a change in the real-world external to the database (through an interface object). Finally all of these definitions are put together to define the REACT object database.

Definition 3.3. Let D be an object database. D is a *REACT object database* if:

a) All object communication is through member function execution and encapsulated events.

b) All reactions to events are implemented using rule objects.

c) All external communication with the real-world is through interface objects.

d) All attributes are defined as either real-time attributes or configuration attributes.

3.2 Encapsulated Events

Traditionally, the only interface to objects is through the invocation of public member functions. Encapsulated events extend the object interface to include events. Just as the algorithms for member function execution are private, event detection algorithms are also private - only the name and description are revealed. This approach greatly simplifies the object programmer's task. He/she is only required to notify the DBMS whenever a public event is detected and is not required to process the event as part of the object's behavior. The DBMS is responsible for maintaining a subscriber list and notifying all subscribers when an event occurs. As an example consider a "stock" object with encapsulated event "price changed". The object programmer would be responsible for triggering the encapsulated event "price changed" when the stock price changes. A rule object that wished to know when the price of this stock changed would simply subscribe to "price changed" and would not have to know all of

the member functions that could possibly change the stock price. The rule object must only be aware of the name of the encapsulated event and its signature.

3.2.1 Advantages of encapsulated events

In this section we list some of the advantages of REACT encapsulated events compared to the way events are typically defined and implemented for a traditional rule base system.

1) Performance. The object programmer has full knowledge of the internal workings of the object and is free to optimize the event detection. In some cases the object programmer can detect an event with minimal computation overhead by placing a trigger statement in the appropriate place.

2) Data hiding and encapsulation is essentially the paradigm that is used for member functions. It has always been the case that the algorithms for performing member functions are private. All publicly accessible member functions must be named and made public. With encapsulated events all algorithms for detecting events are private and all publicly accessible events must be named and made public. This essentially extends the notion of encapsulation to events.

3) Many complex event detection scenarios could require access to private attributes. Encapsulated events are detected internally just as operations are executed internally with no violation of object-oriented principles.

4) With encapsulated events, the rule object programmer simply subscribes to a given event. It is not necessary to know what member functions might trigger that event.

5) Named events, just like named operations, are much easier to use. You specify *what* you want to detect not *how* to detect it. Rule objects only need to know the name of the encapsulated event and its signature.

6) The object programmer is free to completely change the internal data structures and rewrite all of the member functions and event detection. As long as the existing interface stays the same, all users of the object will not be affected in any way.

7) Since rule objects subscribe to a given encapsulated event, the algorithm to fire a rule for that event is faster than the traditional rule base approach.

3.3 Composite Event Detection

With our event model it is possible to create objects to detect higher level events based on a combination of events on other objects. In this case a rule object subscribes to one or more events on other objects and publishes events which are triggered when a combination of events occur on the other objects. As an example, in the pipeline industry it might be desirable to create an object class for detecting pipeline leaks. For a stock trading program it might be desirable to create an object class to detect trends in the market. These objects would then raise higher level events such as "leak detected" or "trend detected" based on a sequence of lower level events such as "pressure changed" or "price changed".

3.4 Rule Objects

With the REACT object database model, the rule programmer defines rules by creating rule classes. The general form of a rule class is given in Figure 3.4. For each application of the rule, one rule object is instantiated with a set of objects as parameters to the constructor. A rule can apply either to a single object, or to an entire class. In the case that a rule applies to an entire class, a rule object is automatically instantiated whenever an object is instantiated for that class. Rule objects subscribe to

```
CLASS:<class-name>

ATTRIBUTES
{<type> <attribute-1>; . . . <type><attribute-N>; }

CONSTRUCTOR(<parameter-1>, . . . , <parameter-N>)
{ <statements> }

PUBLISH
{ <event1>(<parameter-1>, . . . , <parameter-N>) . . .
  <eventM>(<parameter-1>, . . . , <parameter-N>) }

EVENT [<object-key-1>:<event-name-1>]
        (<parameter-1>, . . . , <parameter-N>)
{ <condition and action statements> }

. . .

EVENT [<object-key-M>:<event-name-M>]
        (<parameter-1>, . . . , <parameter-N>)
{ <condition and action statements> }
```

Figure 3.4 General form of a rule class

events on other objects and then perform actions (based on conditions) when notified of an event by the system. The process is described in the following subsections.

3.4.1 Event handlers

The rule programmer must write an event handler for each event that is subscribed to. These event handlers would typically perform actions (based on conditions) when they are executed. The database kernel handles all of the details of event notification, the rule programmer is only responsible for specifying the conditions and actions. As an example a rule object for purchasing would execute member functions on interface objects to buy stock.

3.4.2. Rule conditions

The rule object programmer would be free to call query member functions on other objects as part of condition evaluation. The rule object programmer can not call update member functions as part of condition evaluation. Continuing with the stock example above, the condition to buy stock might be: (price < 100).

3.4.3 Rule actions

The rule programmer has two options for actions: 1) take action directly by calling update member functions on other objects. Or 2) defer actions to another rule object or rule objects by triggering one or more events.

3.5 Summary

The REACT database model provides a powerful environment for implementing real-world active control systems. We will refer to the model described in this chapter as the *REACT database model*. A database system that implements the model will be referred to as a *REACT database system*. A target database created to run on a REACT database system will be referred to as a *REACT database*. The prototype database kernel described in Chapter 4 will be referred to as the *REACT implementation*. We will use this notation unless it is clear from the context. We

describe the implementation of a REACT kernel to implement this model in Chapter 4. Concurrency control algorithms based on this model are described in Chapter 5. We analyze the performance of the REACT database model for single and multiprocessor systems in Chapter 6. Chapter 7 looks at techniques for testing the correctness of target databases. In Chapter 8 a recovery algorithm is given that works for this model and Chapter 9 briefly looks at additional features that are needed for a full blown system. A complete description of how the REACT database model can be used to implement control systems is given in Chapter 10.

CHAPTER 4. PROOF OF CONCEPT PROTOTYPE

An active object-oriented real-time database kernel based on the REACT database model has been developed to test our research concepts. The encapsulated event / rule object model along with the standard member function interface has been fully implemented. The source code for this implementation is given in Appendix C. The resulting REACT database system is a memory resident database designed to support the transaction loads required for the real-time applications described in Chapter 1. The kernel processes incoming transactions and handles all event notifications. In keeping with the REACT model, the only external interface to the database is through member function execution. Each transaction must contain one or more member function execution statements. Each member function statement contains an object-key, method-id, and parameters. After each member function is executed the return value (if any) and all of the output parameters are placed in the output queue to be returned to the calling process. If there is no return value or output parameters a simple success message is returned. If the object-key is invalid, the method-id is invalid, or the parameters are incorrect, an error code along with a message is returned. All real-time transactions are operations on one or more explicitly identified objects.

4.1 Real-time Transaction Execution

All transactions consist of one or more member functions execution statements. As these transactions are executed, encapsulated events could be triggered and must be queued up to be delivered to subscribers. We will need a queue for incoming transactions, a queue for responses, and a queue for triggered events. The basic system diagram to implement this is shown in Figure 4.1. The three main queues used by the REACT Kernel will be: the transaction queue, the response queue, and the event queue. Transactions enter the transaction queue and are processed in arrival order. Each transaction is made up of one or more operations on objects. As a transaction is executed, one or more encapsulated events may be triggered. As each

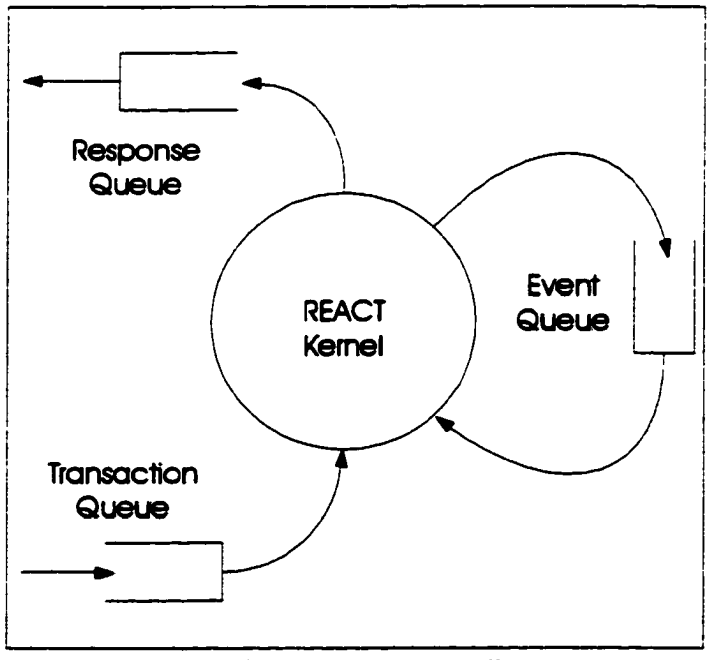

Figure 4.1 The basic REACT system diagram

one of these encapsulated events is triggered, one entry is placed in the pending encapsulated event queue. After the transaction has finished execution, a response is placed in the response queue and, for each entry in the pending encapsulated event queue, an event notification is delivered to all of the rule objects that have subscribed to that event. It is possible that the actions performed on the delivery of event notifications will in turn trigger additional events, which would result in more entries in the pending encapsulated event queue.

4.2 Event Manager

We felt that it was very important that the object programmer should not be burdened with managing subscribers or delivering event notifications. This is important because of the need to simplify the object programers task, to be certain that all event notifications are delivered correctly, and because a database cannot be considered active if the user is required to implement the active functionality. For these reasons, we implemented an event manager to handle all of the details of subscribing for events and delivering event notifications. The REACT event manager maintains subscriber lists for all objects in the database and handles all event notifications when an event is triggered. When an object is instantiated the event manager creates an array of linked lists, one list for each published event (see Figure 4.2). These lists are initially empty and one item is added to a list every time an object subscribes to one of the published

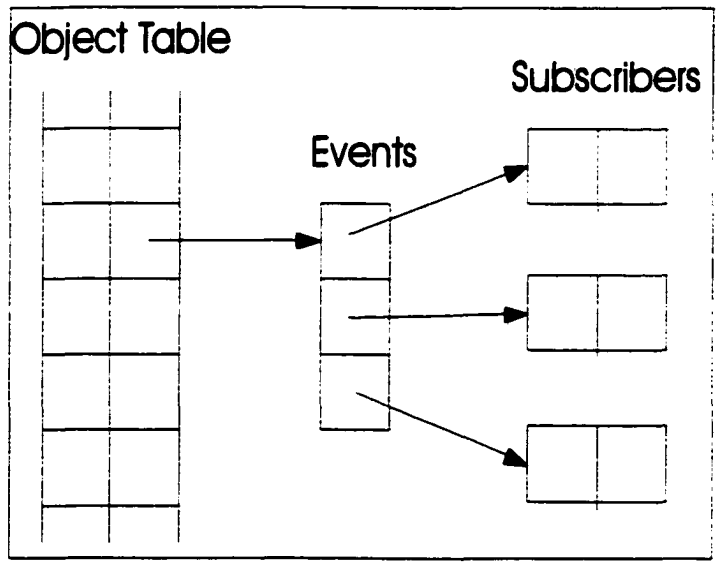

Figure 4.2 The event list structure

events (see the code fragment for subscribing to an event in Figure 4.3). There will be one such set of lists created by the REACT event manager for each object in the database that publishes events. When an object triggers an event the event manager places one entry in the pending event queue that contains the OID (Object ID) of the event originator and the event number. This operation runs in constant time (see the code fragment in Figure 4.4). After the current operation completes there will be zero or more events on the pending event list. For each pending event, the originator OID, which for REACT is the offset in the object table, is used to index into the object table and locate the entry for that object. The event number, which is determined at compile time and passed as an argument, is then used to index into the event array and locate the event list. All of the subscribers on the event list are delivered an event notification. The code fragment to dispatch event notifications for all pending events is shown in

```
event_id_t db_object_t::subscribe_event(char *key, char *event_name)
{
  return oodb->subscribe_event(oid, key, event_name);
}


event_id_t object_database_t::subscribe_event(object_id_t subscriber,
                          char *key,
                          char *event)
{
  db_object_t *obj;
  event_list_t *elist;
  char **events;
  int n_events;

  obj_list->get(key, &obj, &elist);

  if ((obj == NULL) || (elist == NULL))
  {
    return 0;
  }

  short event_number = -1;
  n_events = obj->published_events(&events);

  for (int i=0; i < n_events; i++)
  {
    if (0 == strcmpi(events[i], event))
    {
      event_number = i;
      break;
    }
  }

  if (event_number == -1)
  {
    return 0;
  }

  event_id_t eid = next_event_id();
  elist->add_subscriber(event_number, subscriber, eid);
  return eid;
}
```

Figure 4.3 Code fragments to subscribe to an event

```
void db_object_t::trigger_event(short event_number)
{
  oodb->trigger_event(oid, event_number);
}

void object_database_t::trigger_event(object_id_t event_originator,
                  short event_number)
{
  pending_events[num_pending_events].event_originator = event_originator;
  pending_events[num_pending_events].event_number = event_number;
  num_pending_events++;
}
```

Figure 4.4 Code fragments to trigger event

Figure 4.5. The complexity of this algorithm is constant time for each rule that is fired and O(n) to fire all rules, where n is the total number of rules fired by a transaction. This is the best that can be done for any implementation since there will always be some work to do for each rule that is fired. For the prototype, we have implemented only the deferred coupling mode since there is some consensus [15] that multiple coupling modes need not be supported with real-time databases.

```
void object_database_t::trigger_pending_events(void)
{
  for (int i=0; i < num_pending_events; i++)
  {
    db_object_t *obj;
    event_list_t *elist;

    obj_list->get(pending_events[i].event_originator, &obj, &elist);

    if (elist == NULL)
    {
      continue;
    }

    event_list_elem_t *elem;

    for (elem = elist->list[pending_events[i].event_number];
         elem != NULL;
         elem = elem->next)
    {
      db_object_t *obj;
      event_list_t *elist;
      obj_list->get(elem->subscriber, &obj, &elist);
      if (obj == NULL)
      {
        continue;
      }
      obj->event_occurred(elem->eid);
    }
  }
  num_pending_events = 0;
}
```

Figure 4.5 Code fragment to fire all rules that have subscribed to pending events

4.3 Example Transaction

Our example transaction is an object update that triggers one event for which there is one subscriber. This subscriber executes a member function on another object upon receiving the event notification. Executing this member function then triggers a second event for which there are no subscribers. A breakdown of all the operations executed by REACT when such a transaction is received is given in Figure 4.6. Multiple method invocations for a single transaction are allowed.

1) The object key is hashed to locate the correct object.

2) The generic execute member function is called on that object to locate the member function and verify the arguments.

3) Call and execute the actual method, which also triggers one event.

4) When the object triggers the event the event manager retrieves the OID for the subscriber and uses this to locate the event list for this object. One element is added to the pending event list for the one entry on the subscriber list for this event..

5) When the member function returns, all pending events are dispatched. In this case there is only one event on the list.

6) The event notification is delivered to the subscriber. This subscriber executes some statements after receiving event notification and calls a member function on another object.

7) The member function then executes some statements and triggers one final event for which there is no subscriber.

8) The DBMS checks the event list one more time and finds no more events and proceeds with the next transaction.

Figure 4.6 An example transaction

4.4 Comparison to a Rule Base

The REACT rule system using rule objects is much more efficient than a traditional rule base system. All overhead for the REACT rule system is constant time for each rule that is fired. For a non-indexed rule base the rule overhead when executing a member function is O(n), where n is the number of rules for a given object table. With a traditional rule base implementation, after each update member function is executed, the entire rule base for that object table must be searched. With REACT, rule objects register for named events on specific objects and the overhead is constant time for each rule that is fired. Even with a tree-indexed rule base the overhead would be O(log(n)). The primary reason for the efficient dispatching of rules in the REACT database is that the traditional rule base approach assumes that any rule could possibly be triggered by updating any object. REACT uses the subscription information provided by the encapsulated event system to determine exactly which rules are fired by each object event. For real-time systems this advantage is critical.

4.5 Summary

The main purpose of the kernel implementation was to show that the REACT database model could be efficiently implemented. We were able to implement the REACT rule system such that the rule system overhead is constant time for each rule

fired, which is the best that can be done. This also serves as a test bed for determining how long it will take to execute transactions.

CHAPTER 5. CONCURRENCY CONTROL AND DISTRIBUTED DATABASE ALGORITHMS

In this chapter we consider two ways to execute transactions concurrently: 1) Execute transactions concurrently on a multiprocessor computer or 2) Execute transactions concurrently on separate, networked computers.

5.1 Concurrency Control

Up until this point, we have assumed that all database transactions are executed one at a time (serially). This will be satisfactory for many applications that run on single-processor, memory-resident systems. However, for performance reasons, there are many applications that will require transactions to execute concurrently using multi-processor systems. As is well documented in the literature [33, 97], if we execute transactions in parallel, we must resolve the problems of lost update, uncommitted dependency, inconsistent analysis, and deadlock. We want to make the transactions appear to be atomic. With the memory resident features of REACT and a recovery algorithm like the one discussed in Chapter 8, we are able to support very high transaction loads (on the order of thousands of transactions per second). These high transaction loads create new challenges for concurrency control algorithms. Almost all previous real-time concurrency control research is for database systems that

handle less that 100 transactions per second [1, 52, 79, 84, 86, 91], many of these support less than 10 transactions per second. Such research typically ignores the time that it takes to manage locks, detect deadlock, run scheduling algorithms, etc. With the transaction loads supported by a REACT database system, the time to detect deadlock, the time required for every transaction to support rollback, and the time to run scheduling algorithms becomes very significant. If at all possible, we should prevent deadlock because of the overhead and the unpredictability associated with it. In this chapter we present a concurrency control algorithm for the REACT database system which takes very little time to execute and prevents deadlock.

5.1.1 Background

The key features of REACT and the applications for which is was designed that affect the design of concurrency control algorithms are: 1) real-time, 2) memory-resident, 3) object-oriented, 4) high transaction load, 5) very short transaction times, and 6) rare configuration changes. There has been a great deal of research for real-time concurrency control algorithms for real-time databases [1, 2, 46, 52, 79, 84, 91, 93, 98], but none of this research takes into account all of these factors, especially a high transaction load. Based on the properties of REACT, we have designed a pessimistic concurrency control algorithm that takes very little time to execute, does not rollback executing transactions, and still provides a high level of concurrency for most practical databases. We did not consider optimistic concurrency control

algorithms because they require rollback and because of the unpredictability and overhead associated with rollback. The real value of the REACT concurrency control algorithm is achieved in multiple processor systems. With these systems, we can very significantly increase the throughput by the use of concurrency control algorithms. We will now discuss the features of a REACT database system and typical applications one at a time and how they affect concurrency control algorithms.

Memory Resident. Because target applications will require REACT to be memory-resident, transaction execution times are much shorter and there is no blocking for IO. With disk-based concurrency control algorithms, the time to execute concurrency control algorithms is insignificant in relation to even a single disk access. In contrast, with a memory-resident database, the time to execute concurrency control algorithms is very significant.

Object-oriented. With object-oriented databases such as REACT that fully support encapsulation, operations are not performed by the DBMS, but are performed by individual objects when member functions or event handlers are executed. This essentially defines all of the possible operations and gives us information that can be used to pre-process the database..

Simple transactions. For the target applications the majority of transactions are simple sensor updates to keep the database in sync with the state of the real-world. The need to rollback is reduced since it may take, on average, less time to let transactions run to completion. This, combined with the fact that target applications require REACT to be memory-resident, means the time that a concurrency control algorithm takes to execute becomes extremely significant.

High transaction load. For a REACT system, we assume on the order of thousands of transactions per second. This means that the concurrency control algorithm will have to be executed on the order of thousands of times per second.

Rare configuration changes. Because configuration changes are rare in target applications, we can optimize concurrency control for real-time transactions. Also, pre-processing the database becomes feasible when configuration changes are rare.

5.1.2 Design goals

In this section we start by giving the basic design goals the REACT concurrency control algorithm.

1) If possible we should pre-process the database because of the extra information available from the object oriented-features and because configuration changes are rare.

2) Because REACT is memory-resident and there is a large volume of simple transactions, any concurrency control algorithm will have to run in an absolute minimum amount of time.

3) Because REACT is real-time and there is a large volume of simple transactions, transaction rollback should be avoided. Also adding rollback capability would increase average transaction times because of the extra work to save the original state. We also did not want to burden the object programmer with having to take rollback into consideration when defining object behavior.

5.1.3 Execution groups

With a relational database, there is no restriction on the transactions that can be performed on the database. Even though there are numerous transactions that "do not make sense" and would never be executed on a relational database, the DBMS has to be prepared to execute any and all possible transactions. In contrast, with an object oriented database that fully supports encapsulation, transactions are limited to operations that can be performed by member functions. As it turns out, this added information plus the encapsulated event / rule object features will allow us to pre-process the database to determine which objects can have transactions executed on them concurrently. We discovered that we could pre-determine objects that cannot have transactions executing on them concurrently and group them together in

"execution groups". These execution groups then make it possible to quickly determine if incoming transactions can be executed concurrently. We will now present our work starting with the definitions for query relation set, update relation set, and execution group.

The *query relation set* of an object, QRS(O), is recursively defined as the set of objects that query or are queried by that object. Similarly, the *update relation set* of an object, URS(O), is recursively defined as the set of objects that update or are updated by that object. The update relation set of O and the query relation set of O will always include O itself. We will split the REACT database into N execution groups as defined below.

Two objects O_A, O_B, will be in the same *execution group* G, if and only if:

$$(URS(O_A) \cup QRS(O_A)) \cap (URS(O_B) \cup QRS(O_B)) \neq \varnothing$$

For every database transaction there will be an *execution set* which is the set of execution groups computed by the taking the union of all of the execution groups of the objects that are specified in the transaction. Transactions are *compatible transactions* if the intersection of their execution sets is empty. We extend the definition of update relation set and query relation set to transactions. The *update relation set of a transaction* URS(T) is the union of the update relation sets of the

objects that are operated on by that transaction. The *query relation set of a transaction* QRS(T) is the union of the query relation sets of the objects that are operated on by that transaction.

Lemma 5.1. For a REACT database D, compatible transactions can execute concurrently and the final state of the database will be the same as if the transactions were executed serially.

Proof. Assume that for two compatible transactions, T_1, T_2, executed concurrently, there is a different state than would be obtained by executing them serially. There must be some object that is changed by one transaction and also queried or changed by the other transaction. More formally there is some object O such that:

$$(O \in URS(T_1) \text{ AND } O \in QRS(T_2)) \text{ OR } (O \in QRS(T_2) \text{ AND } O \in URS(T_1)) \text{ OR } (O \in URS(T_1) \text{ AND } O \in URS(T_2))$$

which implies that for two objects O_1, operated on by T_1 and O_2 ,operated on by T_2,

$$(URS(O_1) \cup QRS(O_1)) \cap (URS(O_2) \cup QRS(O_2)) \neq \varnothing$$

which implies that O_1 and O_2 are in the same execution group,

which implies that T_1 and T_2 are not compatible transactions and we have reached a contradiction. ■

In the next subsection we will give an example database and show how it can be divided into execution groups. These execution groups are then used for the REACT concurrency control algorithm.

5.1.4 Example

We will now describe the REACT concurrency control algorithm by first giving an example. We have created an example database with relationships between objects as shown in Figure 5.1. This database is much smaller than any real REACT database would be and is for illustration purposes only. Encapsulated events / rule objects, query member functions, and update member functions have been included for completeness. This database has eight execution groups as shown in Figure 5.2. To create execution groups, each object in the database is visited. We search for objects that are queried by, are updated by, update, or query each object. These objects are then recursively searched, adding each to the current execution group, until we reach objects that have already been included in this execution group or objects that are not updated or queried and do not query or update other objects. In the example, we start

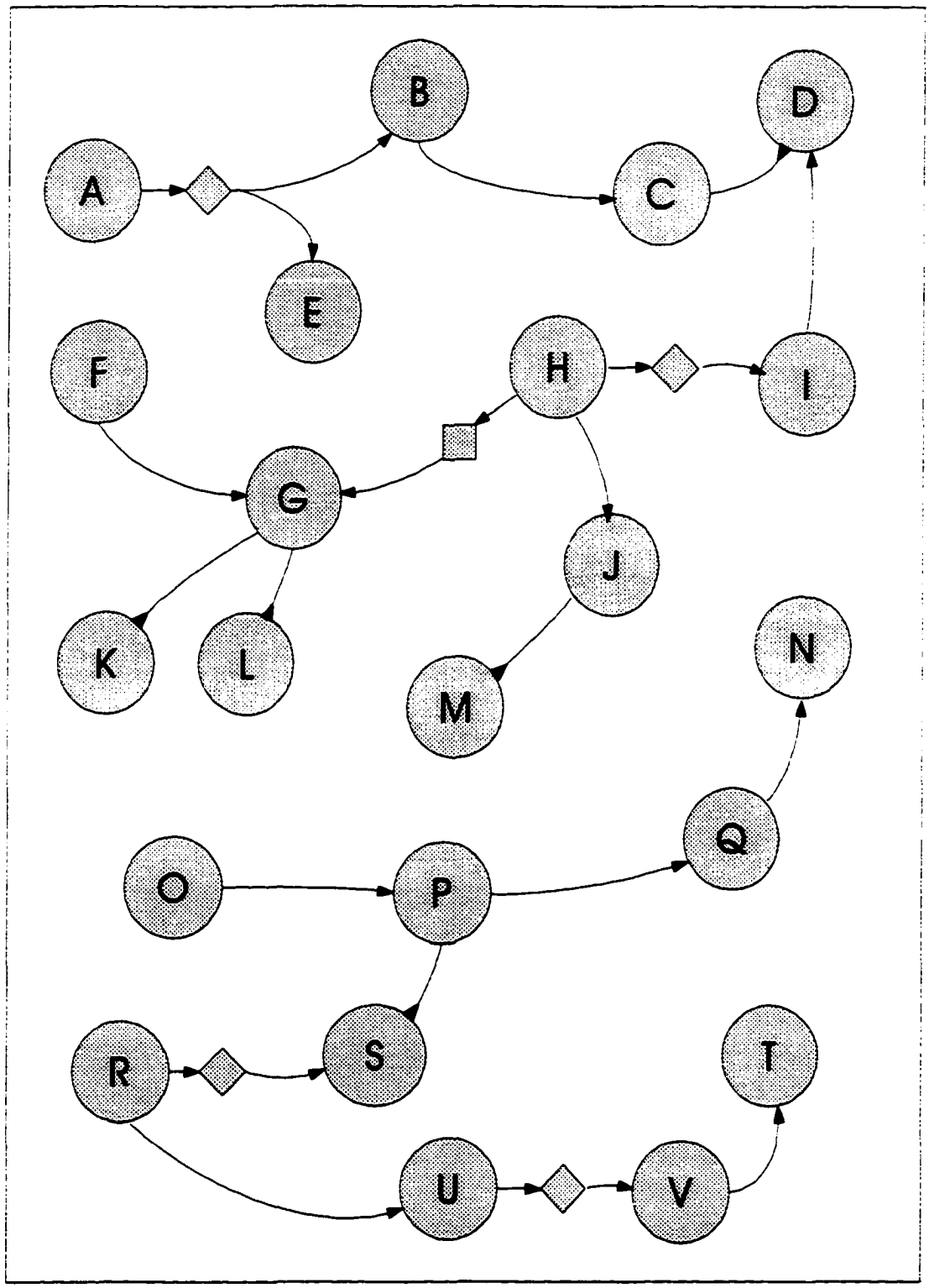

Figure 5.1 Example database

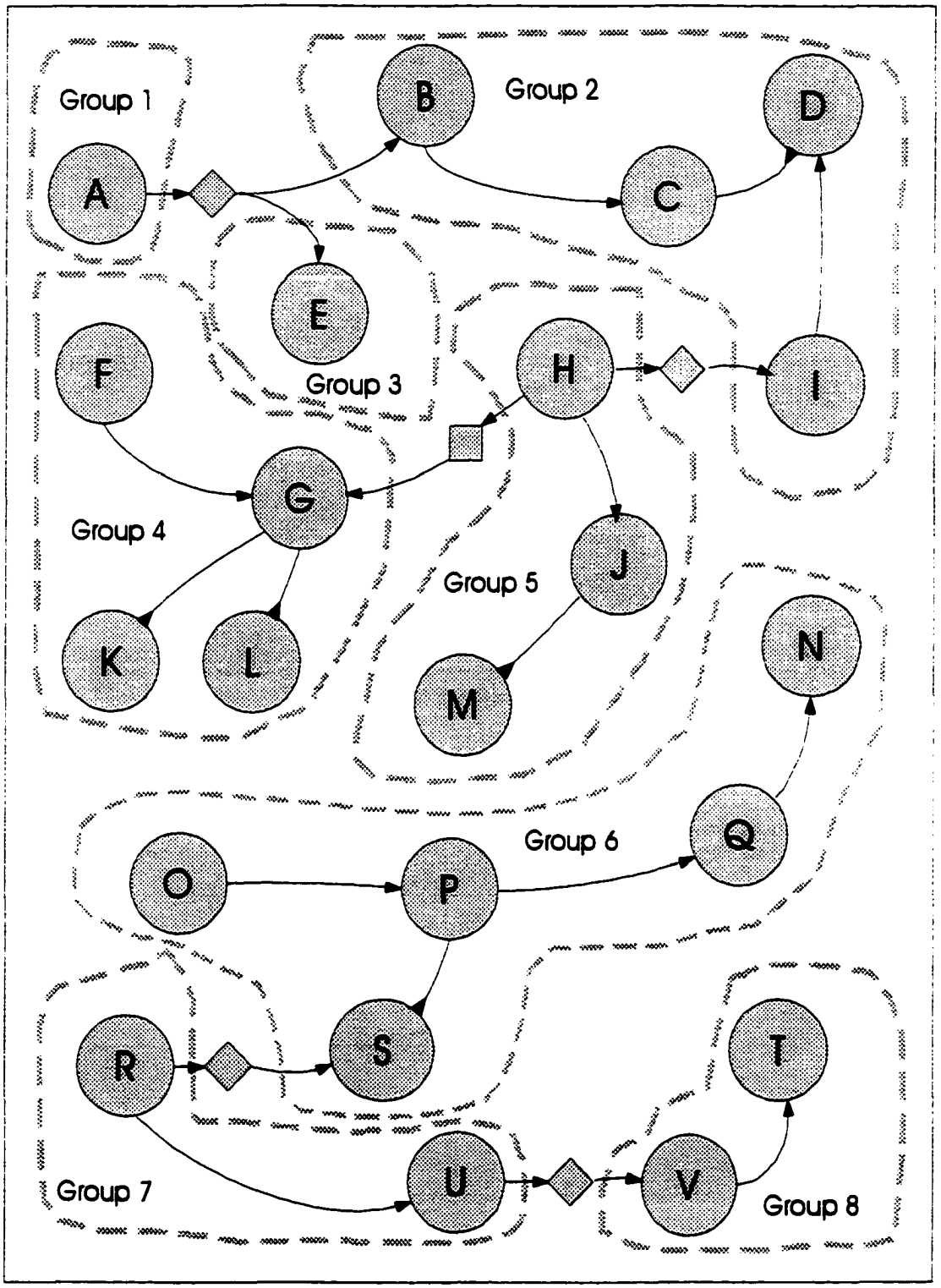

Figure 5.2 Execution groups

with object A. There are no query or update arcs to or from A, so it is in an execution group by itself (Group 1). We move on to object B and find that it updates C, C queries D, and D is updated by I. Group 2 is therefore made up of B, C, D, and I. We skip C and D because they have already been added to an execution group, and move on to E. As with A, there are not any query or update arcs to or from E, so E is in a group by itself (Group 3). F queries G, G queries K, and G also queries L creating Group 4 which contains F, G, K, and L. Groups 5 through 8 are determined similarly. The results are shown in Figure 5.2.

5.1.5 Algorithm

Transactions can be performed concurrently as long as they operate on objects in different execution groups (Lemma 5.1). The basic algorithm is then only to allow a transaction to execute as long it is compatible with all currently executing transactions. We must also limit the total number of concurrently running transactions. More formally, the algorithm is as follows:

1) A transaction is eligible for execution if it is compatible with all currently executing transactions and it is compatible with all transactions waiting to execute.

2) Eligible transactions are considered for execution in order of arrival.

3) Eligible transactions will be started as long as the total number of transactions currently executing is less than the number of processors.

These rules are reevaluated for all waiting transactions when a new transaction arrives or a transaction completes execution. Rule 1 guarantees that no incompatible transactions will execute at the same time and that lower priority transactions cannot execute until all waiting incompatible higher priority transactions have executed. Rule 2 guarantees that eligible transactions that arrived first will be considered first. Rule 3 above prevents us from starting too many transactions. First, the throughput is not increased by starting more transactions than processors, since there is no blocking for IO. Second, the overhead of context switches will adversely affect performance.

This is essentially a conservative, locking algorithm where each transaction locks all of the objects in each of its execution groups before it is allowed to execute. Any transaction that contains one of the same execution groups is then blocked until this transaction completes. The granularity of this algorithm will be dependent on the number of execution groups in the target database. Notice that all resources are allocated at once so that transactions run to completion without blocking once they are started. We will now prove that no two non-serializable transactions will execute at the same time, and that deadlock can not happen for the REACT concurrency control algorithm.

Lemma 5.2. The REACT concurrency control algorithm will guarantee that no two non-serializable transactions will execute at the same time.

Proof. From Rule 1, no transaction will be scheduled to execute unless it is compatible with all currently running transactions. From Lemma 5.1, all compatible transactions are serializable. ■

Lemma 5.3. With the REACT concurrency control algorithm, deadlock will never occur.

Proof. For deadlock to occur there are four necessary conditions [69, 75]: 1) mutual exclusion, 2) hold and wait, 3) no preemption, and 4) circular wait. With the REACT concurrency control algorithm, all resources for transactions are allocated at once and then transactions run to completion with no waiting, so the hold and wait condition can never occur. If one of the necessary conditions can never occur, then deadlock can never occur. ■

5.1.6 Concurrency control summary

We have developed a concurrency control algorithm suitable for high transaction loads (thousands of transactions per second). Because we pre-process the database, determining if any two transactions can run currently is reduced to comparing execution groups. This pre-processing is feasible since configuration

changes are rare for the target applications for which the REACT model was designed. In Chapter 6 we simulate the REACT database running on a multiprocessor system using this concurrency control algorithm.

5.2 Distributed Database Algorithms

A second way to achieve concurrent execution of transactions is to distribute objects to separate REACT databases running on separate computers. In this chapter we discuss how we can distribute REACT objects among multiple networked computers.

5.2.1 Objects that must be together

With REACT database systems we require that all transactions run to completion with no blocking. For this reason if one object executes member functions on another we will require both objects to be on the same computer. Our algorithm will not allow objects that execute member functions on each other to be on different computers.

5.2.2 Objects that should be together

Rule objects that subscribe to events on another object should, if possible, be on the same computer. This is not a strict requirement since it is not necessary to block

while event notifications are delivered. If these objects are not on the same computer, then there will be extra overhead and a networking delay when the event notifications are sent to another computer. For this reason our algorithm will attempt to minimize the number of subscribers for an event that are not on the same computer as the publisher for that event.

5.2.3 Distribution algorithm

We will use the execution groups computed in the previous section to determine which objects *must* be on the same computer. All objects in the same execution group must be on the same computer. Also, for each pair of execution groups to be on different computers, we will compute the number of objects that subscribe to events on objects in the other group. The cost function will then be:

Cost = the number event notifications that must be transmitted between databases

This cost function will minimize the number of objects that receive event messages from object on another database. There is extra overhead with these events since they must be sent across the network. The algorithm would then be to evaluate exhaustively all possible splits of approximately the same size that would result in the least cost. The problem is that the complexity of this algorithm would be exponential. If we create a different graph then we can use a heuristic algorithm by He et al. [48] to

solve this problem. The graph we need to create for this algorithm has one node for each execution graph. Arcs connect nodes if and only if an event "connects" the two execution groups. The weight of the arc is the number of events that "connect" the two groups. With this graph the partition can be created using the cluster partition given by He et al [48].

There could also be other criteria such as that all of the objects that represent values from a particular geographic location should be on the same server if possible. The user might want to determine a split manually and then verify that it would be a valid split and the cost associated with it in terms of the number of events that would have to be transmitted between database servers. As an example, we will use the database given in Figure 5.1. If we consider all splits that put approximately half the objects in each database, the least cost split is shown in Figure 5.3. This is the only split that has zero cost. The user might have some other criteria that are more important, such as all objects from a certain geographical area should be in the same database. In this case, the user would manually choose a split, then verify that it is a valid split, and also determine the cost associated with the split.

5.2.4 Distributed database summary

This algorithm would be too expensive to execute if we must exhaustively consider all possible splits and the cost associated with each split. As with concurrency control, this algorithm is possible because of the extra information available from

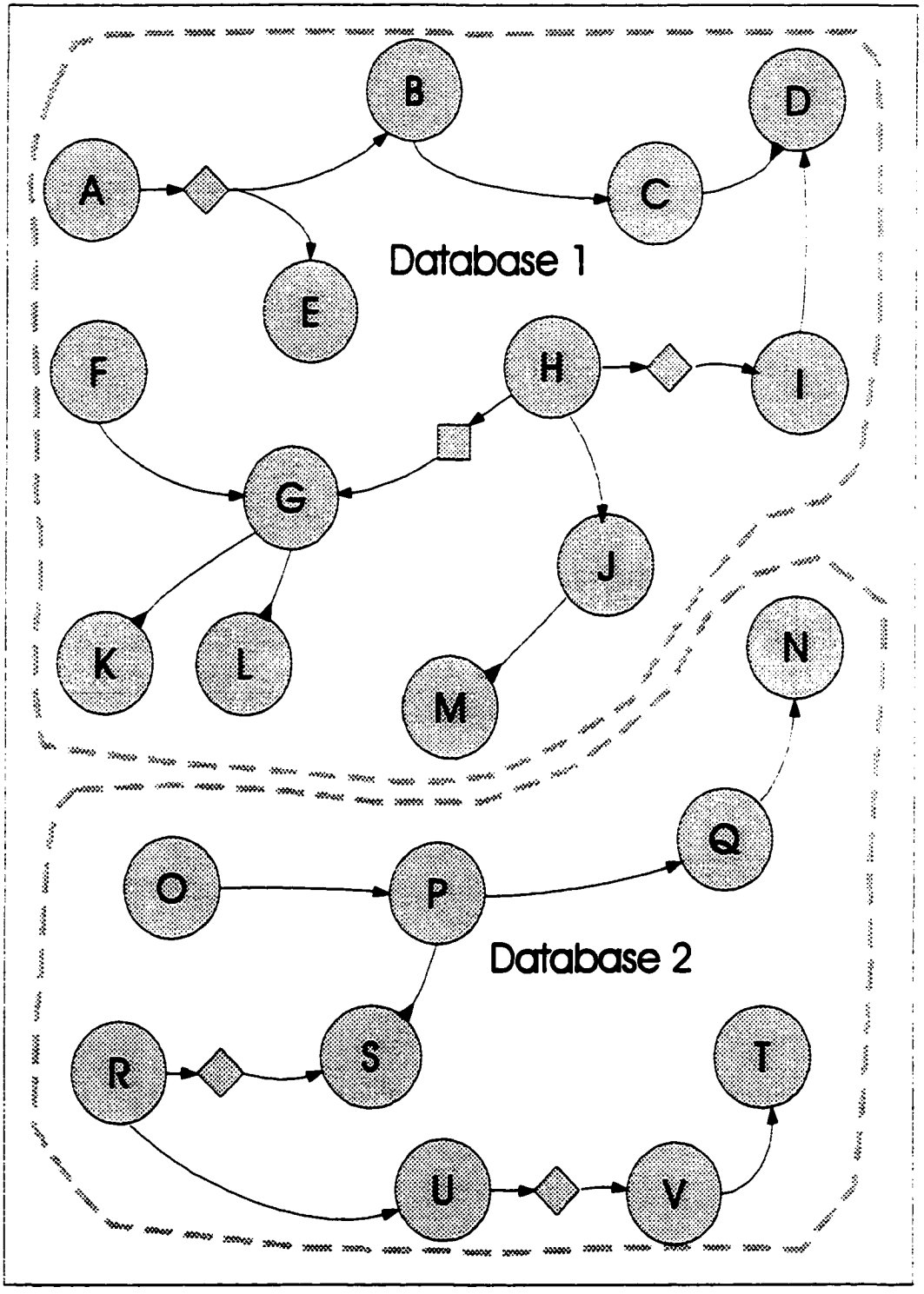

Figure 5.3 Example database split

REACT database model. We can determine which objects operate on each other and require them to be on the same computer. Because of the high transaction loads and predictability required for real-time systems, we do not allow any blocking once a transaction has started to execute.

5.3 Summary

We have given two methods to obtain concurrent execution of transactions for the purpose of increasing performance. It would be possible to use one or both of these methods depending on the performance requirements of a given application.

CHAPTER 6. ANALYSIS

In this chapter, the performance of the REACT database is analyzed for both single and multi-processor systems. For the single-processor case we used a closed-form solution, and for the multi-processor case we use a simulation.

6.1 Background

Any given target real-time application will require a certain amount of processing power in order to meet all deadlines. This amount of required processing power will vary as a function of time for almost all applications. An efficient implementation (such as REACT) will reduce the amount of processing power required but the required processing power will still vary as a function of time. The problem is then to make sure that the required processing power will never exceed the available processing power or to make sure that there is enough processing power such that the probability of exceeding the required processing power is acceptably small. This is somewhat complicated in the multi-processor case since there may be processing power available, but because of concurrency control issues, transactions cannot be scheduled to run in time to meet their deadlines. In some cases it may be possible to put an upper bound on the required processing power. In these cases, the problem is then reduced to obtaining hardware that will provide the maximum

processing power required or possibly splitting the database to run on separate CPU's until the maximum processing power is never exceeded. This splitting is possible with REACT and we have developed algorithms to determine which objects can be placed in separate databases in the event that a split is required (see Chapter 5). For some target real-time applications it may not be possible to put an upper bound on the maximum processing time required. In situations such as these, a statistical approach is the only alternative. The REACT model and rule system will significantly reduce transaction execution times and decrease the probability of a missed deadline. We have done statistical analysis for both single and multi-processor systems. Even in the case that it is possible to determine an upper bound for the required processing power it may still be desirable to use a statistical approach in the case that the cost of providing such processing power would be prohibitive and the chances of using such an amount of processing power would be acceptably small. In this chapter we will not take into consideration things such as round trip transmitter delays, networking delays, operating system latency, etc. All these things would have to be factored into the final analysis. The primary motivation in this chapter is to show that transactions will not miss their deadlines for lack of processing power or because they cannot be scheduled. An upper bound would typically be used to factor in all of the other delays. It might be desirable to use a networking technology, such as token ring, that would provide more deterministic networking delays.

6.2 Single Processor Analysis

For single processor systems, we give examples using an upper bound approach and a statistical example.

6.2.1 Single processor upper bound example

In this section we show an example where an upper bound can be placed on the required processing power to guarantee that deadlines will not be missed. Assume a system with 10,000 objects updated every 20 seconds. The transaction deadline is 10 milliseconds. Assume that a careful examination of all possible transactions for this application indicated that no transaction would take more than 1 millisecond. Such an examination, of course, would be quite tedious and would be dependent on objects created for this application and the REACT database and rule system. In order to even out the loading, all inputs are updated at equal intervals throughout the 20 second scan period. In the worst case we will use 50% of the available CPU time and no deadlines will be missed since all updates happen at fixed intervals.

This method is somewhat restrictive. All inputs must be scanned periodically, no event-driven inputs would be allowed, and only simple pre-defined transactions are allowed. It may also be very tedious to examine all possible transactions to guarantee that they will all complete within a given time. In many cases these disadvantages are outweighed by an absolute guarantee that no deadlines will be missed.

6.2.2 Single processor statistical example

Assume that the inter-arrival times of transactions are exponentially distributed and that service times are exponentially distributed, the probability that any transaction will miss its deadline can be calculated as follows:

$$P = \frac{1}{\exp(T_D\mu(1-\lambda/\mu))}$$

Where: T_D is the deadline for each transaction in seconds, λ is the arrival rate in transactions per second and μ is the service rate in jobs per second. We can also calculate the mean time between missed deadlines (in minutes) as follows:

$$M = \frac{\exp(T_D\mu(1-\lambda/\mu))}{60*\mu}$$

From tests of the prototype of Chapter 4, we found that about 30,000 member functions can be executed every second. It is assumed that there would be multiple member functions executed per transaction and that the operations might be more complex than for our tests. As a conservative estimate based on our simulations of the REACT protype, it is assumed that for typical applications 2,000 transaction could be executed per second. The effect of load on missed deadlines is shown in Figure 6.1 assuming this execution rate and a deadline of 5ms. The effect of execution time on

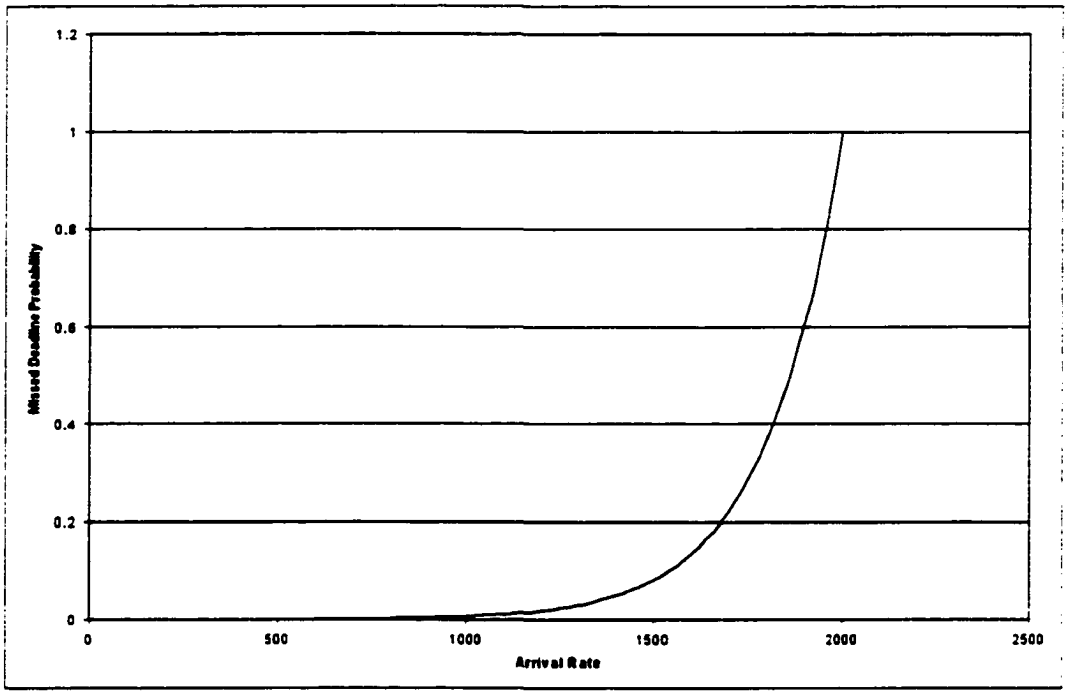

Figure 6.1 Effect of load on missed deadlines

missed deadlines is shown in Figure 6.2 assuming an arrival rate of 2,000 transactions per second and a deadline of 5ms. The parameters used (including the distributions for the arrival rate of transactions and the service times for transactions) will depend on the actual real-time system being implemented. The transaction execution times are also heavily dependent on the REACT database system. The calculations would have to be redone on a case-by-case basis to determine the probability of a missed deadline for a given real-time system implemented using REACT. The purpose of these calculations is to show that for a typical system the probability of a missed deadline using the REACT database can be negligible if the average loading is kept under about 25% and the transaction times are significantly less than the deadline times.

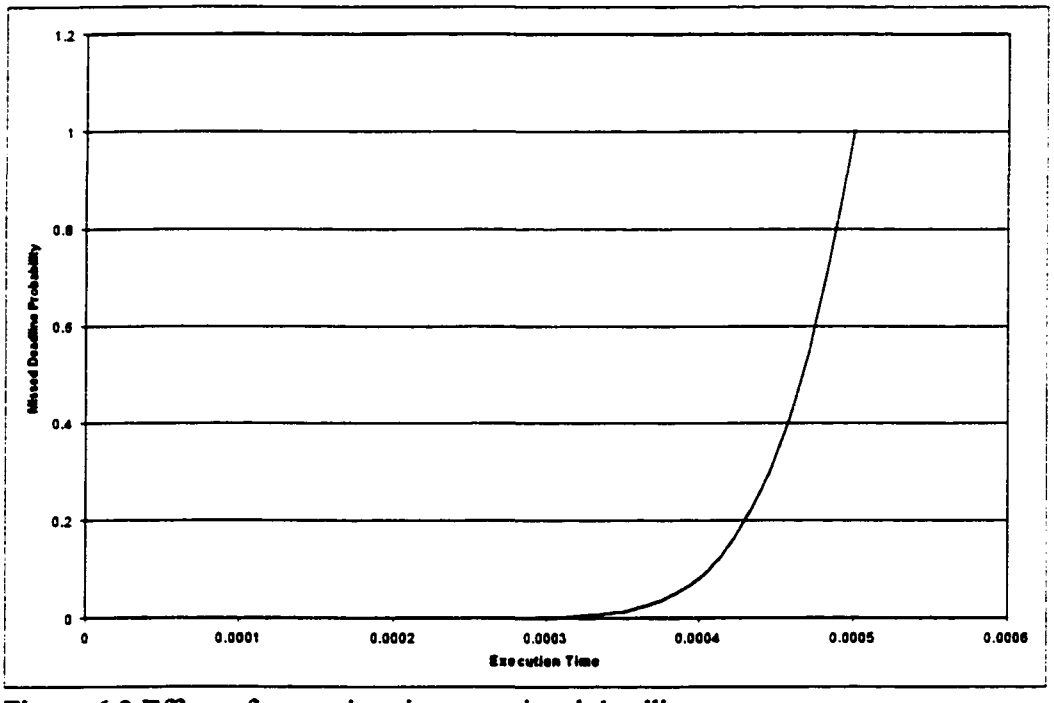

Figure 6.2 Effect of execution time on missed deadlines

In summary, there are two ways to make sure that the probability of missing a deadline is acceptably small: 1) Determine an upper bound for the maximum required processing power and provide that maximum processing power (probability of missing a deadline is zero), or 2) Use a statistical approach based on knowledge of the target application to show that the probability of missing a deadline is acceptably small. It should be noted that no implementation developed for a specific real-time application can provide absolute guarantees that all transactions will meet their deadlines if it is not possible to put an upper bound on the processing power required for that real-time application.

6.3 Multi-Processor Analysis

For the multi-processor case we developed a simulator to simulate the REACT database running on a multiprocessor system using the REACT concurrency control algorithm. In this section the simulator is described and the results of our simulation are given.

6.3.1 Simulation model and assumptions

In this section we describe how the REACT database, rule system, and concurrency control overhead are modeled for the purpose of simulation. The REACT database system is modeled as a single queue system with multiple identical servers. It is further assumed that one processor is reserved to perform concurrency control algorithms and other system functions. The inter-arrival times and the member function execution times as well as the event handler execution times for rules are assumed to be exponentially distributed. It is assumed that all fired rules are started as separate transactions and that these "triggered transactions" are included in the average transaction load of incoming transactions. We assume that the REACT rule system overhead for firing rules is included in transaction times. This is a reasonable approximation since, with the REACT model, the rule overhead is constant time for each rule fired. The cost of dispatching a REACT rule amounts to indexing into an array and following some pointers. This overhead would be well under a microsecond

for a 200 MHz class processor and insignificant in relationship to the time to execute a member function (assumed to be over 100 microseconds for a typical application). For other rule systems such as those that require a search of the rule base, the rule system overhead would have to be accounted for separately as a function of the total number of rules. It is assumed that transactions are executed first come first serve except that transactions are only allowed to execute if they are compatible with all executing transactions and are compatible with all transactions that arrived earlier as required by the REACT concurrency control algorithm. When a transaction is generated by the simulator, it is randomly determined if that transaction is compatible with each other transaction in the system, based on the probability of two transactions being compatible, as described below in the discussion of the random number generator. This compatibility information is stored and used whenever two transactions need to be tested for compatibility. We model the REACT concurrency control overhead as a fixed amount of overhead that is incurred for each pair of execution groups for each transaction in the system when a transaction arrives. It is assumed that the concurrency control algorithm must be performed sequentially on incoming transactions before they are eligible for execution and that this time is added to the total time in the system.

The random number generator used is a multiplicative linear-congruential generator given in *The Art of Computer Systems Performance Analysis* [55], page 443. The generator was tested using the test values given to verify a correct implementation. Separate random number streams using different seeds were used for

each simulation parameter. To generate an exponentially-distributed random number, first a uniform random number is generated, and then the inverse transformation of the cumulative distribution function of the exponential distribution is used to obtain a number drawn from an exponential distribution [55]. To determine if two transactions are compatible we generate a uniform random number between zero and one. If the this generated number is less than or equal to the probability of an incompatible transaction, the transactions are assumed incompatible, otherwise they are assumed compatible. Each simulation run is for 10,000 transactions, with new random number seeds used after every 1,000 transactions.

The input parameters to each simulation run that can be varied are: transaction arrival rate, member function execution time, average number of member functions per transaction, the probability that any two transaction are incompatible, deadline time for transactions, and concurrency control overhead factor. Our main purpose in varying these parameters was to look at the sensitivity of REACT model to our assumptions. The output of each transaction run is the number of missed deadlines, average processor utilization, and average number of transactions in the system.

6.3.2 Simulation results

The primary purpose of the simulations was to determine the probability of missed deadlines under various operating conditions in a REACT system. We also wanted to determine what probability of incompatible transactions was required to

fully utilize available processors. We assumed eight processors for all of the simulations. The raw output data from the simulations are given in Appendix E. The base parameters used for simulations are given in Table 6.1. These are the parameters used for simulations unless otherwise noted.

Table 6.1 Base simulation parameters

Parameter	Value	Units
Arrival Rate	12000	Transactions/sec.
Execution Rate (single processor)	8000	Functions/sec.
Average # of Member Functions	4	
Probability Incompatible	0.05	
Transaction Deadline	4	ms
Concurrency Overhead	10	ns

For the first experiment we varied the arrival rate of transactions for various probabilities of incompatible transactions (Figure 6.3). For each probability, we plot load vs. missed deadlines. As expected there is a degradation in performance with an increasing probability of incompatible transactions. The reason for this degradation is that with higher probabilities of incompatible transactions, processor utilization goes down because transactions must wait for incompatible transactions that have already started executing even though processors are available. The results of this effect are summarized in Figure 6.4 where we plot maximum average processor utilization vs. arrival rate. The maximum average processor utilization was determined for each probability by increasing the load until the queue started to overflow.

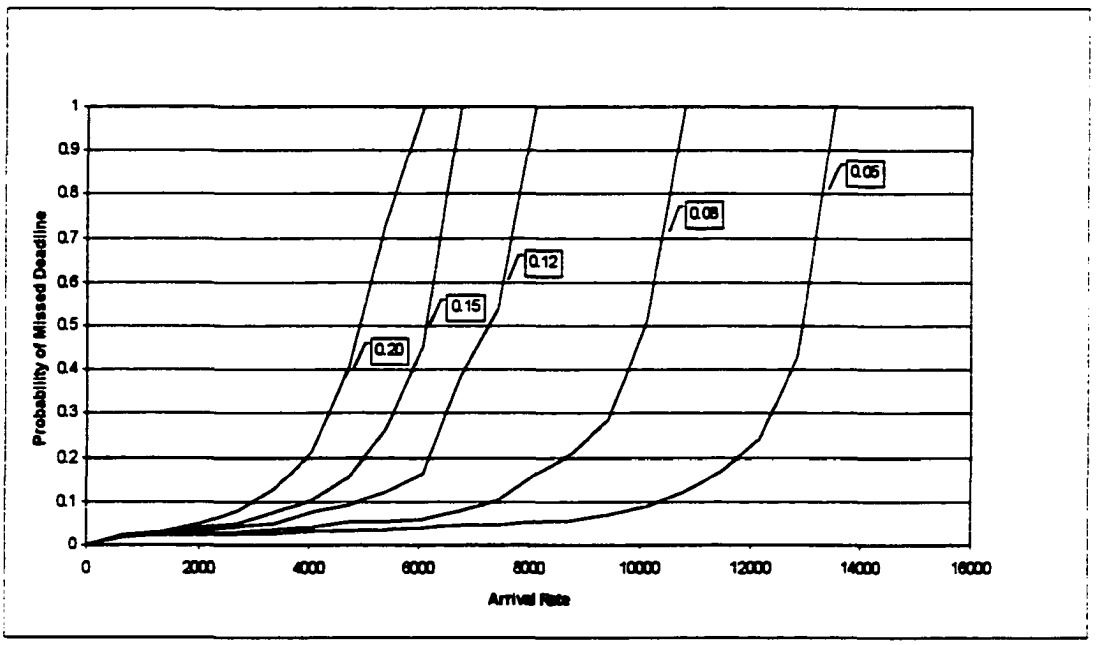

Figure 6.3 Effect of incompatible transaction probability on missed deadlines

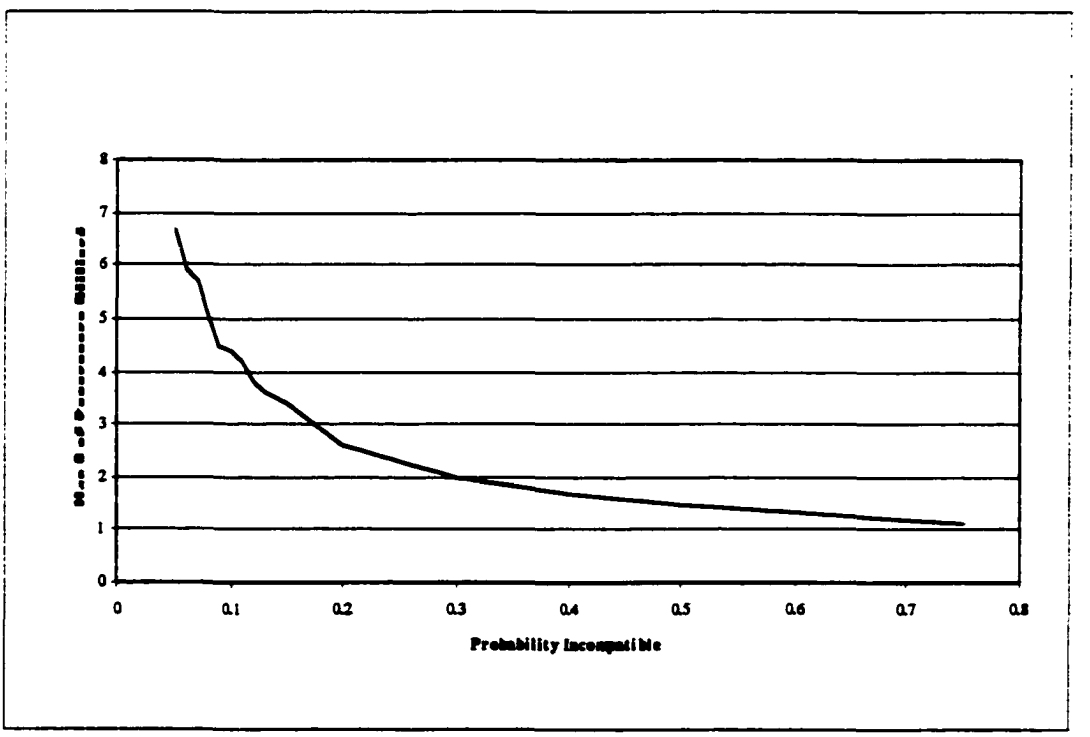

Figure 6.4 Processor utilization as a function of probability incompatible

For the second experiment we varied the arrival rate of transactions for various deadline times to determine the effect of slack time on missed deadlines (Figure 6.5). As expected, the probability of a missed deadline goes up with less slack time. With deadlines of 1 ms there was not sufficient slack to have an acceptable probability of missed deadlines at any load. As noted by others [1], poor performance can be expected if there is not adequate slack time.

Finally, for the last experiment we varied the transaction time to show the effect of increased processor power on missed deadlines (Figure 6.6). If the processing power were doubled so that we could execute 16,000 transactions per second instead

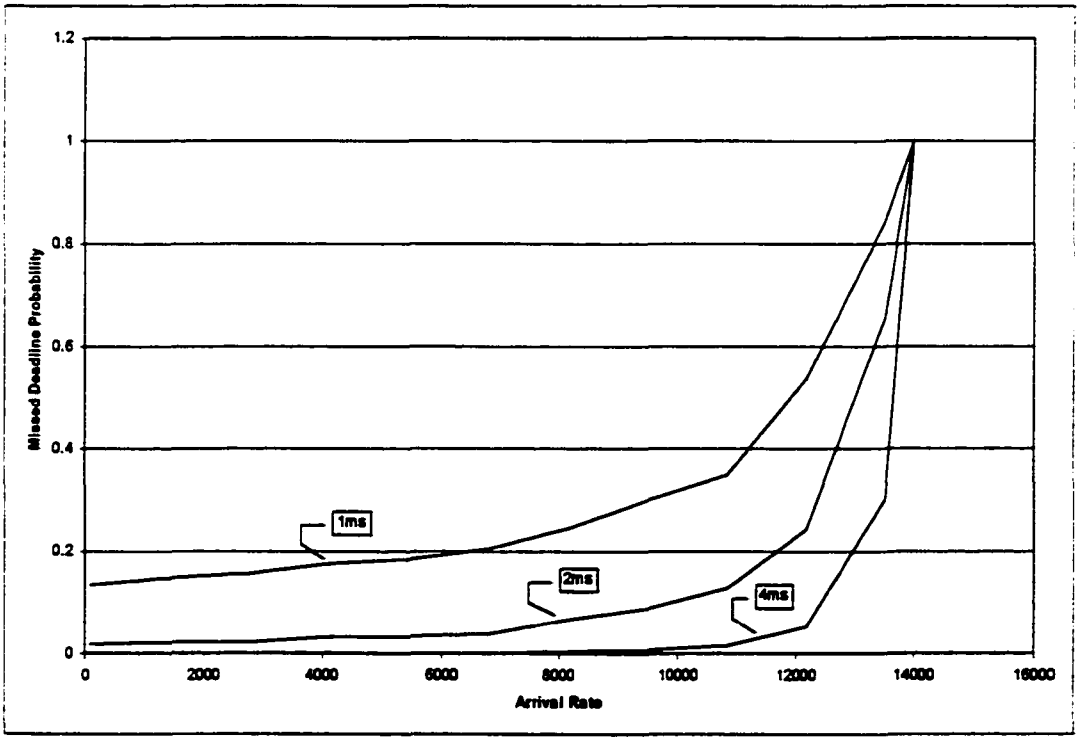

Figure 6.5 Effect of slack time on missed deadlines

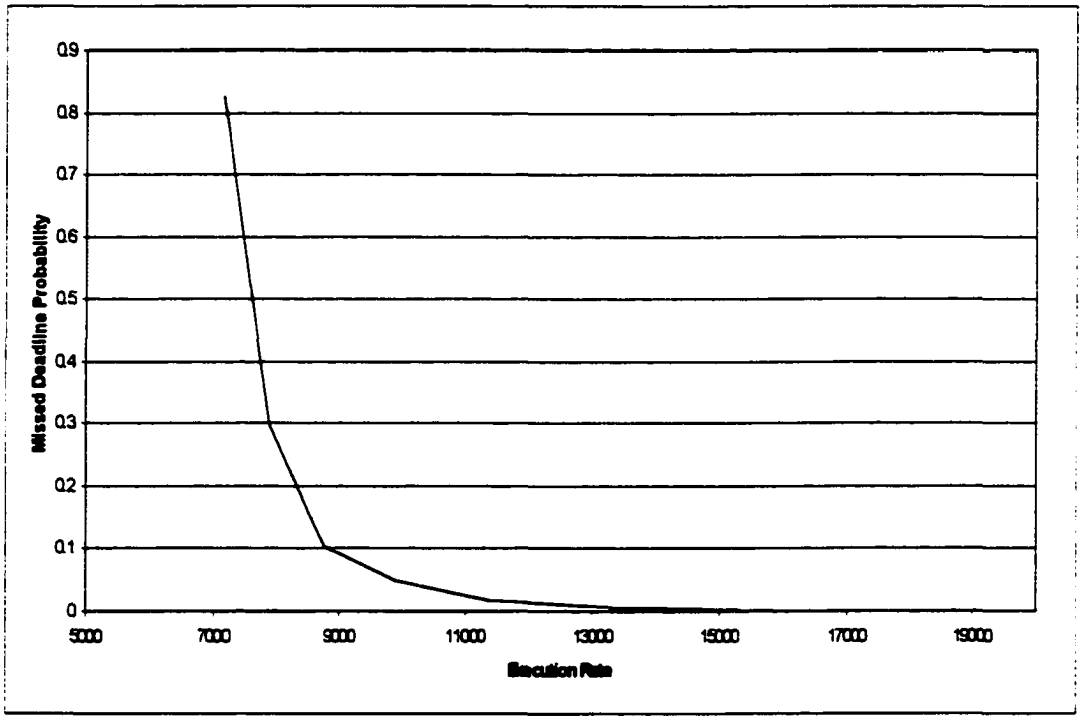

Figure 6.6 Effect of processing power on missed deadlines

of 8,000 transactions per second we could go from over 20% missed deadlines to a negligible probability of missed deadlines. Also, as we saturate the processors and approach the maximum system throughput, the probability of missed deadlines goes up very sharply.

6.3.3 Simulation summary

The probability of incompatible transactions has an effect on the maximum number of processors that can be utilized. In order to fully utilize the processors on an eight processor system the probability of incompatible transactions needs to be less than 0.05. We have looked at an application for a natural gas distribution system and

estimated that it would have about 3,000 objects if implemented using REACT and would have over 500 execution groups[1]. This would be typical of large control systems for which REACT is designed and would allow for a more than adequate amount of concurrency for four or eight processor systems. With 500 execution groups, the probability of any two transactions being incompatible will be about 0.002 assuming that the execution groups are about the same size. Almost all applications could expect to have a probability of missed deadlines of less than 0.05. In order to have an acceptable probability of missed deadlines, there must also be adequate slack time. The probability of a missed deadline is over 15% for even light loads when the transaction deadlines are 1 ms and there is very little slack time. The deadlines, loads, and acceptable levels of missed deadlines will vary from application to application. This simulator would have to be re-run for each application to determine the probability of missed deadlines for the associated operating parameters. This simulator is a powerful tool for assessing the performance of REACT for a particular application. It also gives us insight into how REACT performs under various operating conditions and how to achieve a negligible probability of missed deadlines.

---

[1]The data was acquired through a confidential consulting contract.

CHAPTER 7. TARGET DATABASE CORRECTNESS

In this section we look at a set of static analysis techniques for the REACT database model. Similar work [8, 9] has been done for traditional rule bases on relational databases. With the REACT database model, member functions are executed or events are triggered automatically when an event is detected internally by an object and the necessary conditions are met. With real-time systems, it is very important that certain properties hold so that we can guarantee deterministic behavior. For the REACT database model we consider the following three properties:

Property 1. The property *termination* holds if it can be guaranteed that all processing will terminate after any operation on the database.

Property 2. The property *confluence* holds if the final state of the database will be the same, regardless of the order in which rules are fired.

Property 3. The property *observable determinism* holds if the order of operations performed on the real-world will always happen in the same order, regardless of the order in which rules are fired.

It can be difficult or impossible to determine if these properties do or do not hold for a given instance of an active database. We have, however, developed conservative algorithms for the REACT active database model that will: 1) Guarantee that all operations for a REACT database will terminate or say that they may not terminate, 2) Guarantee that the set of rules objects for a REACT database is confluent or say that the set may not be confluent, and 3) Guarantee that the set of rule objects for a REACT database is observably deterministic or say that the set may not be observably deterministic. Since the algorithms are conservative, the error, if one exists, is in saying that a property does not hold when in fact it does.

Also, when these algorithms find a problem, and are not able to guarantee any one of the three properties, they will isolate the set of objects that are responsible for the problem. The rule programer may then be able to fix the problem and re-run these algorithms. These tools could be used interactively in this way to guarantee all three properties hold for a REACT database.

7.1 Termination Analysis

For any database, since the number of computations contained in the database is finite, there must be some cycle of computations if any operation on the database does not complete. In this section we describe how computation cycles occur with member functions, encapsulated events, and rule objects in a REACT database. There

are two kinds of cycles that can occur in a REACT database: 1) Cycles that are internal to an individual object (loops and recursion), and 2) Cycles that involve multiple objects. Since examining code for infinite loops is beyond the scope of this paper, we will assume that all operations on individual objects terminate. To address cycles involving multiple objects we show how the execution of code in one object can result in the execution of code in another object for the REACT model. For a relational database the only cycles possible are from the active features of the database (rules). Because REACT is object-oriented, cycles are also possible from the standard object-oriented features. We first consider the kinds of cycles that arise from the standard object-oriented features of REACT and then we will discuss the kinds of cycles that are introduced as a result of the active features of REACT (rule objects).

For the standard object-oriented features of the REACT database model there are only two ways that the execution of code in one object can cause the execution of code in another object: 1) calling query member functions (one object queries another), and 2) calling update member functions (one object updates another) A *query cycle* is any set of objects $O_1..O_N$ that form a cycle such that: O_1 queries O_2, O_2 queries O_3, . . . , O_{N-1} queries O_N, O_N queries O_1. Figure 7.1 shows a query cycle formed by the objects B, C, D. An *update cycle* is any set of objects $O_1..O_N$ that form a cycle such that: O_1 updates O_2, O_2 updates O_3, . . . , O_{N-1} updates O_N, O_N updates O_1. Figure 7.2 shows an update cycle formed by the objects B, C, D. Notice that a cycle

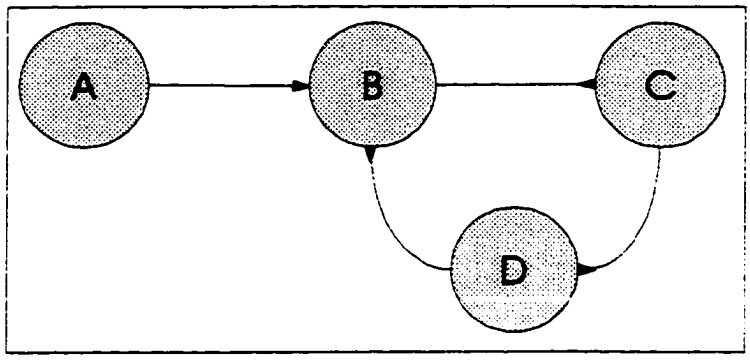

Figure 7.1 A query cycle

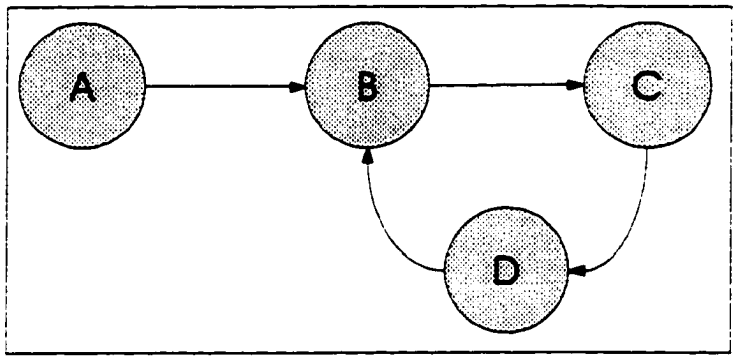

Figure 7.2 An update cycle

that contains both queries and updates as shown if Figure 7.3 will always terminate, because query member functions can never call update member functions by definition.

Because of the active features of the REACT database model, we have introduced another way that the execution of code in one object can cause the execution of code in another object. With REACT, if an object triggers an event, this can cause the execution of code in a rule object that has subscribed to this event. This is indirect in that the object notifies the DBMS kernel of the event occurrence, and the DBMS kernel then fires all rule objects that have subscribed to this event. Even though this in indirect, we will say that one object *fires* another object (a rule) in this fashion.

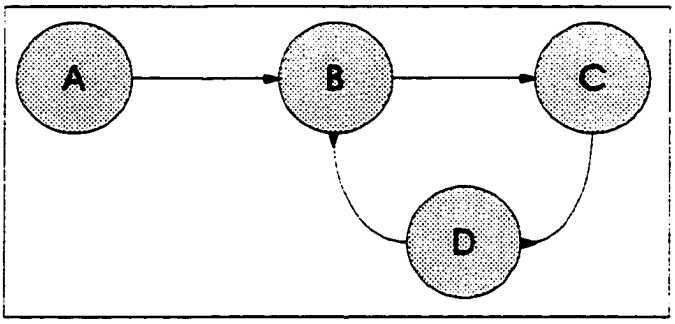

Figure 7.3 Terminating cycle

Also since an event handler for a rule object can call update member functions and update member functions can trigger events, we can have a cycle that is a combination of updates and fires defined as follows. A *rule cycle* is any set of objects $O_1..O_N$ that form a cycle such that: O_1 fires or updates O_2, O_2 fires or updates O_3, . . . , O_{N-1} fires or updates O_N, O_N fires or updates O_1. A rule cycle is shown in Figure 7.4.

To find these three kinds of cycles in a REACT database, we create an execution graph where each node is an object and there are three directed edge types: query, update, and fire. The problem is then to search the execution graph for query, update, and rule cycles. Finding cycles in a REACT database is then reduced to finding

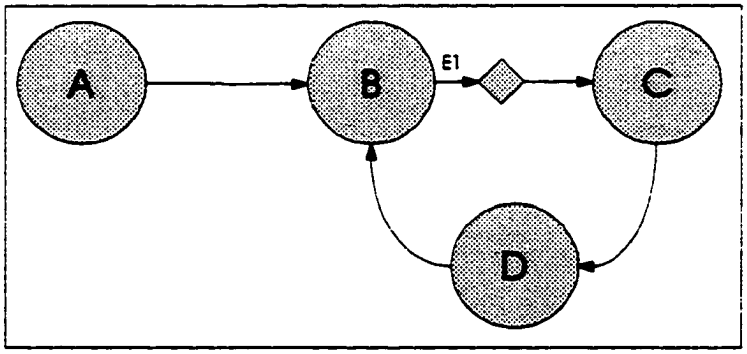

Figure 7.4 A rule cycle

cycles in a directed graph. This problem has been solved many times. As an example we refer you to *Data Structures and Algorithms* by Aho, Hopcroft, and Ullman [7], page 221. The algorithm they give uses the depth first search where we look for back arcs at each node.

Termination algorithm.

1) Build a graph with update, query, and fire edges as described previously.

2) Search for update, query, and rule cycles using the algorithm given in *Data Structures and Algorithms* [7].

3) Print all such cycles that are found.

Lemma 7.1. (Termination) For a REACT database D, any operation on D will terminate if no cycles are found by the termination algorithm.

Proof. Assume that some computation does not terminate. Since the number of computation blocks in the database is finite, there must exist some cycle of computations in the database. Since we take as an assumption that all individual operations terminate, the only possible cycles involve multiple objects. The only possible cycles for the REACT database model involving multiple objects are query cycles, update cycles, and rule cycles as described previously. Any such cycles would have been found by the graph algorithm given in Data Structures and Algorithms [7]

and we have reached a contradiction. We refer you to Data Structures and Algorithms [7], page 221, for a proof of the algorithm for finding cycles in a directed graph. ■

The complexity for this algorithm will be O(n+e) where n is the number of objects and e is the number of edges. The pseudo code for this algorithm is given in Appendix A. Java code for this algorithm is given in Appendix B. We use algorithm for finding cycles in a directed graph given in Data Structures and Algorithms [7].

7.2 Confluence Analysis

When an object is operated on (update member function execution, query member function execution, rule object event handler execution), there will be a set of objects that can be queried and a set of objects that can be updated as a result of operating on that object. The *query set* of an object (QS) is the set of objects that can be queried as a result of operating on that object. The *change set* of an object (CS) is the set of objects that can be changed as a result of operating on that object. When an event is triggered there is a set of rule objects that have subscribed to this event and are fired by the DBMS. The *rule set* of an event RS(E) is the set of rule objects $R_1..R_N$ that are directly fired when E is triggered. Simply put, rule objects are interfering rules, when one rule object changes or queries an object that is changed by the other rule

object. This of course would result in unpredictable behavior of the database since the order of rule firing is not defined. Now a more rigorous definition will be given.

Definition 4. Assume some event E with rule set RS(E) = $\{R_1..R_N\}$. Two rule objects, R_i and R_j, both elements of RS(E), are *interfering rules* if:

$CS(R_i) \cap (CS(R_j) \cup QS(R_j)) \neq \varnothing$ or $CS(R_j) \cap (CS(R_i) \cup QS(R_i)) \neq \varnothing$.

Interfering handlers are shown in Figure 7.5. Next it will be shown that we can change the order of any two consecutively fired rule objects and the final state of a REACT database will remain unchanged after any operation, if there are no interfering rules. This will then be used to show that the property confluence holds for a REACT database if there are no interfering rules.

Lemma 7.2. For a REACT database D, assume that some order of firing has been chosen for all rule sets $R_1..R_N$ in D such that R_i is fired before R_{i+1}. For any operation on D, the final state of D will be the same after we change the order of firing for any two consecutively fired rule objects, R_i and R_{i+1}, fired by some encapsulated event if there are no interfering rules in D.

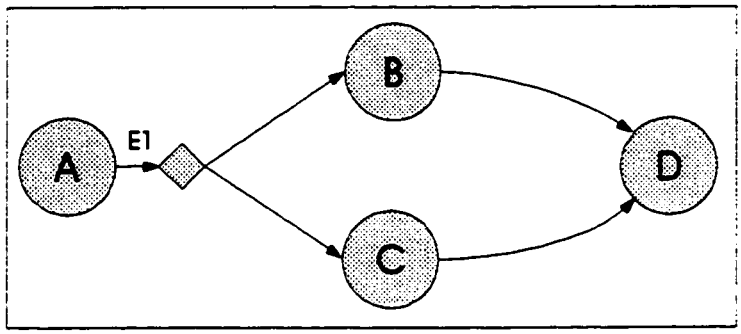

Figure 7.5 Interfering handlers

Proof. Assume that for some operation on the database the final state of the database is different after changing the order of firing for two consecutively fired rule objects, R_i and R_{i+1}. The only way that there could be a difference is if there is some object O such that:

$$(O \in QS(R_i) \text{ AND } O \in CS(R_{i+1})) \text{ OR } (O \in QS(R_{i+1}) \text{ AND } O \in CS(R_i)) \text{ OR}$$
$$(O \in CS(R_i) \text{ AND } O \in CS(R_{i+1}))$$

Which implies:

$$CS(R_i) \cap (CS(R_{i+1}) \cup QS(R_{i+1})) \neq \varnothing \text{ OR } CS(R_{i+1}) \cap (CS(R_i) \cup QS(R_i)) \neq \varnothing.$$

Which implies R_i and R_{i+1} are interfering rules and we have reached a contradiction. ■

We will now use Lemma 7.2 to prove Lemma 7.3.

Lemma 7.3. (Confluence) For a REACT database D, after all operations on D, the final state of the database will be the same regardless of the order of rule firing for any rule sets, if there are no interfering rules.

proof: We can change the order of firing for any two consecutively fired rule objects in a REACT database without changing the final state of the database if there are no interfering rules (Lemma 7.2) . Since all possible orders of firing of all rule sets can be achieved by changing the order of two consecutively fired rule objects at a time, the order of rule firing will never affect the final state of the database after any operation.

■

We will now give the confluence algorithm. The basic algorithm is to visit every object in the database checking for interfering rules. Lemma 7.3 tells us that if there are no interfering rules, then confluence holds.

For every object in the database:

1) Determine the update set and query set using the depth first search algorithm for each rule object in each rule set of this object.

2) Test every pair of rules in each rule set to see if they are interfering rules using the definition.

3) Print all interfering rules that are found.

The worst case complexity of this algorithm will be O(n(n+e)) where n is the number of nodes and e is the number of edges. We will now show that this algorithm guarantees confluence.

Lemma 7.4. For a REACT database D, the property confluence holds if no interfering rules are found by the confluence algorithm.

proof: This algorithm visits all rules in the database and checks for interfering rules using the definition. From Lemma 7.3, if the database does not contain any interfering rules then the property confluence holds. ■

Again, this is a simple graph algorithm using the same execution graph given for termination. The query set of a rule object is found by recursively following all query edges, adding each object we encounter to the query set. The change set of a rule object is found by recursively following all update edges and fire edges, adding each object we encounter to the change set. We then check that $CS(R_i) \cap (CS(R_j) \cup QS(R_j)) = \varnothing$ and $CS(R_j) \cap (CS(R_i) \cup QS(R_i)) = \varnothing$, for all rule objects, R_i and R_j, both members of the same rule set. If this is not true then we have found interfering rules. The query sets and change sets are found by using the depth first search that is common in the literature. As an example, see the depth first search algorithm given in *Data Structures and Algorithms* [7], page 215. This algorithm has been implemented

in Java (Appendix B). Every time interfering rules are found, the object and event are printed along with the two interfering rule objects.

7.3 Observable Determinism Analysis

We now define conflicting rules. Basically conflicting rules are two rule objects, members of the same rule set, that cause observable actions. This could result in unpredictable behavior since the order of rule object firing is not defined.

Definition 5. Assume two rule objects, R_1 and R_2, both elements of the rule set. R_1 and R_2 are *conflicting rules* if they can both cause observable actions.

An example of conflicting handlers are shown in Figure 7.6. It will now be shown that the property of observable determinism holds for a REACT database if

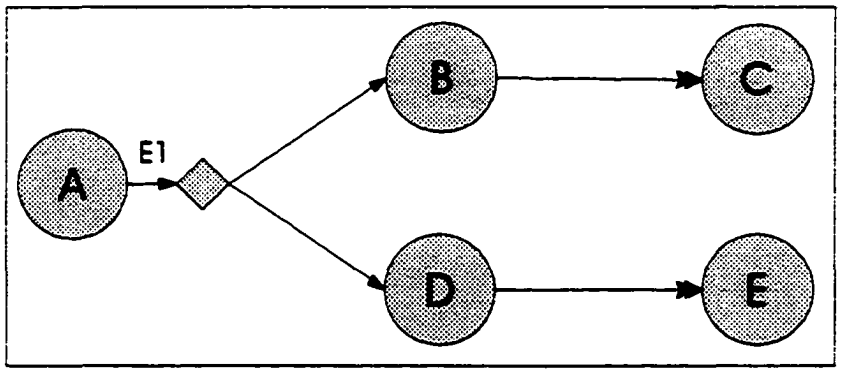

Figure 7.6 Conflicting handlers

there are no conflicting rules. The basic idea is that two rules objects, fired by the same event, should not cause changes to the real world (observable actions).

Lemma 7.5. (Observable Determinism) Assume a REACT database D, if there are no conflicting handlers in D, then D is observably deterministic.

proof: Assume D is not observably deterministic. Since D is not observably deterministic then for some rule set there is a different order in observable actions taken. Since each object operation must always result in the same order of actions, there must be two rule objects in the same rule set that perform observable actions. By definition, such a pair of handlers are conflicting handlers and we have reached a contradiction. ■

We will now give the observable determinism algorithm. The basic idea is to visit every object in the database searching for interfering rules.

Observable determinism algorithm.

For every object in the database:

1) Determine if each rule object for each rule set of this object can cause observable actions.

2) Check each rule set for conflicting handlers using the definition.

3) Print all rule sets with conflicting handlers that are found.

Lemma 7.6. For a REACT database D, the property, observable determinism, holds if no conflicting handlers are found by the observable determinism algorithm.

proof: This algorithm visits all rule objects in the database and checks for conflicting handlers using the definition. From Lemma 7.5, if the database does not contain any conflicting handlers then the property observable determinism holds. ■

The worst case complexity for this algorithm will be O(n(n+e)). To implement this algorithm we add a new edge type to our execution graph (this is shown as a double headed arrow in figure 7.6). This new edge type is for member functions on interface objects that cause actions in the real-world. To determine if a rule object can cause observable actions, we recursively follow all update and fire links looking for calls to member functions that cause observable actions. As soon as we find one observable action, we stop and mark the original rule object as causing observable actions. We check all of the rules sets in the database to make sure that there are not two objects from the same rule set that can cause observable actions. We use the depth first search as given in the previous section. This algorithm is also implemented in Java (Appendix B). Each time conflicting rules are found, the object and event are printed along with the two conflicting event handlers.

7.4 Summary

In this chapter we give powerful algorithms for determining that the properties termination, confluence, and observable determinism hold. If these properties do not hold, then there is the possibility of non-deterministic behavior when operations are performed on the database. Since the algorithms are conservative, the error, if one exists, is in saying that a property does not hold when in fact it does. When these algorithms find a problem, and are not able to guarantee any one of the three properties, they will isolate the set of objects that are responsible for the problem. The rule object programer may then be able to fix the problem and re-run these algorithms. These tools could be used interactively in this way to guarantee all three properties hold for a REACT database

CHAPTER 8. RECOVERY AND BACKUP

The material in this chapter is presented for completeness. While it is not unique to the REACT model, it focuses on a significant subset of the applications targeted by our work on the REACT model. There needs to be a method for restoring the database to a consistent state after a system crash or hardware failure. Also, in many cases a standby computer that is running a copy of the database and ready to take over, is required so that in the event of a hardware failure in the primary system, a standby system can be up and running again within seconds. We will refer to a standby computer set up to take over in the event of a failure of the primary system as a *hot backup*. The recovery algorithm is complicated by a memory resident database such as REACT and by real-time environments where extended downtime would be catastrophic. This is the case with many of the applications targeted by this research. With the transaction loads expected for a REACT system, it is also not possible to write a log to disk after every transaction. A recovery system has been designed for REACT that guarantees that the system can be restarted in a valid state within seconds of a crash without writing to a log file after each transaction.

When we first started work on REACT, we realized that REACT would have to be memory resident to support on the order of thousands of transactions per second required in the targeted applications. This high transaction load even precludes writing

a log to disk for every transaction, since this would take on the order of milliseconds. Also, for many of the applications for which the REACT model was designed, it is not possible to stop transactions if the database is down for any reason, since we cannot "stop the world". This means that if REACT is down for any reason, transactions will be lost. With many of the applications for which REACT was designed the single most important feature of a recovery algorithm is the ability to be up and running and controlling the system as soon as possible. Although the existence of "holes" in the data due to lost transactions is very undesirable this can be tolerated, especially if this is traded for being up and running very quickly in the event of failure. This means that, unlike systems that handle financial transactions, the type of applications handled in this chapter can tolerate not recovering all transactions in the event of a system crash even though it is very undesirable. This recovery algorithm, trades the ability to completely recover all transactions in the event of a system crash for the ability to handle very high transactions loads and the ability to be up and running within seconds after a crash. As it turns out, this recovery algorithm also makes it possible to support a hot backup and even an offsite backup.

8.1 Real-time and Configuration Transactions

With the applications that REACT has been designed for, inserts, deletions, and configuration changes are very rare. This means that configuration transactions are

also very rare. On the other hand, a REACT database needs to support on the order of thousands of real-time transactions per second. These real-time transactions are for the purpose of keeping the state of the database in step with the state of the real-world. If the system crashes, we can save any pending configuration changes to be applied after the system is restarted. On the other hand, when the system crashes, all real-time transactions are lost until the system is restarted since it is not possible to stop changes that are occurring in the real-world. Even if there were some method of buffering the transactions remotely while the system was down, there wouldn't be enough CPU capacity to process them when the system was restarted.

8.2 Snapshot Algorithm

For our REACT implementation, we have designed a snapshot algorithm to periodically save the real-time attributes to disk (see Figure 8.1). For this algorithm to work all real-time attributes must be stored in a common area. We will refer to this area as the real-time buffer. The DBMS provides special routines to allocate space for real-time attributes in the real-time buffer. When it is time to save the state of the database, transactions must be momentarily stopped to guarantee that the database is in a valid state. When all currently executing transactions have completed, the complete contents of the real-time buffer is copied to another temporary buffer. As soon as this copy has completed, transaction processing can be resumed. As

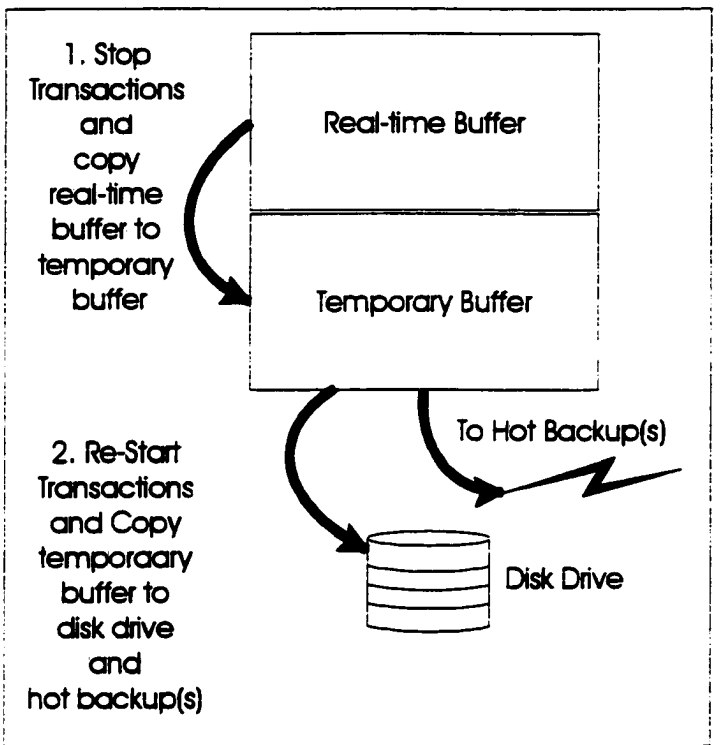

Figure 8.1 REACT snapshot system

transactions continue, the contents of the temporary buffer are copied to a local disk and also to the hot backup computer(s). These snapshots could then be used to restart the primary system in the event of a system crash, or to startup a backup system in the event of a hardware failure on the primary system. With this double buffer algorithm the transactions are stopped for an absolute minimum amount of time. The amount of memory required for this double buffer is also very reasonable. With 10,000 objects and an average of 100 bytes of real-time attributes for each object, we require only 2 megabytes for both the real-time buffer and the temporary buffer. The cost for one megabyte of memory is negligible at today's prices, the time to copy a one megabyte buffer is minimal, and the snapshot is copied to disk and to hot backup(s) by a low

priority background task that is only scheduled when there are no database transactions to execute. If we assume a 200 MHz processor, one instruction per clock cycle, 8 bytes moved per instruction, the time to copy a one megabyte buffer is 625 microseconds. This means that transactions would be stopped for about 625 microseconds while the real-time buffer is copied to the temporary buffer. To minimize the impact, for some applications, this could be scheduled during a time when there are no sensor updates occurring. Also, for a typical system, the average CPU loading from the database would be under 50 percent so there will be plenty of free CPU time to copy the temporary buffer to disk without interfering with real-time transactions.

8.3 Off-Site Backup

This snapshot algorithm also works very well in cases where an off-site backup system is required. In this case it is desirable to have a complete duplicate control system including field communications equipment at a remote location in the case of a fire or explosion at the primary site. In this case, we would have two hot backups, one at the primary site, and one at an off-site location. Snapshots would be copied via modem or other suitable communications device as often as possible from the primary site to the remote site. In the event of a catastrophe at the primary site, there would be a recent valid copy of the database available at the remote site so that the remote site could be up and running and used to operate the control system in a matter of seconds.

We are not aware of any other main-memory database recovery algorithms that would work over a low bandwidth communications device such as a modem.

8.4 Summary

This recovery algorithm works for high transaction loads where it is not possible to write to a log file after each transaction. It will not work for applications that require complete recovery of all transactions up to the point of failure. This algorithm is effective for applications with very high transaction loads where transactions are lost while the system is down and the goal is to minimize total transaction loss. The total transaction loss is the sum of the transactions lost before the system went down plus the transactions lost while the system is down. Many of the target applications run continuously, and cannot be stopped since the real-world cannot be stopped. As a result this loss of transactions is critical in these applications. As noted earlier in the chapter, this work is not unique to the REACT model and has been included for completeness. The control problems in the set of target applications would have the characteristics which this approach is based on. The discussion of using REACT in a control environment given in Chapter 10 assumes this approach to backup and recovery.

CHAPTER 9. ADDITIONAL REACT FEATURES

Most real-world applications will require other features such as a history server, an alarm manager, and a time keeper. A complete discussion of these features is beyond the scope of this research and the basic features are discussed here only for completeness. The example system given in Chapter 10 makes use of the features described in this chapter. After these features and additions to the object interface are discussed, the complete REACT object interface that includes all of these features is given. In order to reduce the load and increase predictability, the history server and alarm manager would not reside on the same server as the REACT database. The time keeper would have to reside on the same server since it generates timer events for REACT objects.

9.1 History Server

Most large real-time systems will require some form of history storage. As an example, a pipeline company might sell gas to a municipality and need to measure the flow of gas to that municipality. The pipeline company will need to bill the municipality on a monthly basis for gas used. To do this we need to have some notion of history, it is not enough to know just the current flow. It would not be feasible to

implement history as part of the REACT database because of the immense storage requirements and other special requirements. Also, it is critical to minimize the additional loads placed on a REACT database system, in order that we can minimize the chance of missing real-time deadlines. In this chapter, only one type of history will be described (floating point history). This is the most common type of history, although some applications might require others.

9.1.1 Floating point history

In this section we will describe a floating point history server that accepts and summarizes/compresses history for items that can be represented by a floating point value.

9.1.1.1 History storage requirements. Typical applications for which REACT was designed might require an enormous amount of history. As an example, we assume history on 1,000 floating point values that are updated every 15 seconds and are represented by a four byte floating point number. For this example, 23 megabytes of data are generated ever day, 691 megabytes of data are generated every month, and 8 gigabytes of data are generated every year. This assumes a very efficient storage algorithm. With a standard relational database a timestamp and index would be used, requiring as much as five times more storage space, on the order of 40 gigabytes per year. Besides taking up enormous amounts of disk space, data in this format are of

little use. We would not likely need to know the flow in a pipeline at a specific date and time, rather we would typically want to know what was the total flow for a particular hour, week, or month. In the next subsection an example of compressing and summarizing history is given.

9.1.1.2 History compression/summarization. The most common type of compression/summarization used is to store the average value observed over a period of time rather than all of the values observed. As an example, by summarizing the data into monthly averages, we can reduce the storage requirements for a years worth of data from 8 gigabytes to 48 kilobytes. For a typical system we might want to store all history values observed for the last 24 hours, daily averages for the last month, and monthly averages indefinitely. With a one gigabyte disk, we could store over 20 years of history in this manner. Also, the data are stored in a format that is the most efficient for typical queries.

9.1.1.3 History API. To add floating point history to the REACT database we would need to add API (application programing interface) calls to the base database object class for reporting history, for creating history, and deleting history as shown in Figure 9.1.

```
int dbPoint::createHistory(char *description);
dbPoint::recordHistory(int historyNumber, float historyValue, time_t
time);
dbPoint::deleteHistory();
```

Figure 9.1 Example history API

The create history member function creates history for this object and returns a handle for use in accessing the history. The recordHistory member function is called whenever history should be recorded. The deleteHistory member function is called to delete history.

9.1.2 Other history types

For completeness we mention that other kinds of history may be required for some systems. For example, in might be necessary to keep history on items that can be represented by a boolean value or an integer value. We might also want to keep a history of operator actions, so that in the event of a failure we can analyze the reactions of operators. These other history types could possibly require different kinds of storage and compression / summarization.

9.2 Alarm Server

For REACT most events that are generated will be handled by rule objects. There will however be some events that cannot be handled by objects and require

human intervention. Examples of such events might be "LeakDetected" or "ValveFailed". In both cases humans will have to intervene to handle these events. A database rule cannot repair leaks or fix valves. We define *database alarms* as special events that require human intervention. A subsystem will be required to keep track of all alarms and display them to operators.

9.2.1 Alarm summaries types

We define three kinds of alarm summaries that would meet the needs of most large control systems: 1) an *active alarm summary* is a list of currently active alarms, 2) an *unacknowledged alarm summary* is a list of alarms that have occurred but have not been acknowledged, and 3) a *historical alarm summary* is a historical listing of all alarms that have occurred.

9.2.2 Alarm record

A standard alarm record would be required for use by the alarm manager to display information and status to operator. The required attributes would vary from application to application. A proposed alarm record that would meet the needs of most applications is given in Figure 9.2.

```
struct alarmRecord
{
  char objectKey[25];          /* Object that detected this alarm */
  char objectDescription[40];/* Description of this object */
  int alarmNumber;            /* Identifies which alarm for this object*/
  char alarmDescription[40]; /* Description of this alarm */
  long alarmID;               /* Unique ID for this alarm */
  int priority;               /* Priority of this alarm */
  time_t time;                /* Time this alarm occurred */
  int severity;               /* Caution, Alarm */
  int locationCode;           /* Physical location of this alarm */
  boolean currentlyActive;    /* Is alarm currently active */
  boolean acknowledged;       /* Has this alarm been acknowledged? */
  time_t acknowledgeTime;     /* Time this alarm was acknowledged */
  char acknowledgedBy[25];    /* User who acknowledged the alarm */
};
```

Figure 9.2 Alarm record

9.2.3 Alarm API

There needs to be an API for objects to communicate the alarm state of an object to the alarm manager. Objects would be required to notify the alarm manager when an alarm becomes active, an active alarm becomes inactive, and the severity of an alarm changes. The only information the object programmer must provide is the alarm number, alarm description, and alarm severity, all other information is provided by the base database class or determined by the alarm manager, simplifying the alarm API considerably. The base class will provide the object key, object description, priority, and location code. All other information is determined by the alarm manager. The REACT alarm API is given in Figure 9.3.

```
void activateAlarm(int alarmNumber, char alarmDescription[], int
severity);
changeAlarmSeverity(int AlarmNumber, int newSeverity);
void deactivateAlarm(int alarmNumber);
```

Figure 9.3 Proposed alarm API

9.3 REACT Timekeeper

An important aspect of real-time systems is the time at which actions are performed. In some cases it is sufficient to perform actions at the time that data arrive. In this case, spontaneous events are generated as data arrive from the real-world to trigger multiple actions. In many control and monitoring applications this simple model will not be enough. As an example, for some control applications we will need to perform real-world actions at regular intervals that do not necessarily coincide with the random arrival of data. For monitoring applications, we will often need to give real-world systems time to react after changes in output before we check them for validity. *Periodic timers* generate events that happen at regular intervals starting at some initial start time. *One-shot timers* generate one event that happens at a given time after some initial start time. Timer events will be treated the same as encapsulated events. There will be a pre-defined timer object on every system that publishes periodic and one-shot timer events. The object programmer then subscribes to timer events like any other event.

9.4 The REACT Object Interface

In this Section, we will tie together all of the features to describe the complete database object interface proposed for REACT. The complete definition of a REACT object will include attributes, private member functions, constructors, query member functions, update member functions, encapsulated events, event handlers, alarms, and history as shown in Figure 9.4. We will separate the interface into the private parts, accessible only by the object programmer, and the public parts, available to the users of objects.

9.4.1 Public object interface

The public REACT object interface is a set constructors, a set of query member functions, a set of update member functions, a set of published events, a set of event handlers, a set of alarms, and a set of history items. This is the public object interface that is available for each REACT object and can be obtained by querying the REACT database.

9.4.2 Private object interface

The private interface that is available only to the object programmer will include, in addition to the public interface, a set of configuration attributes, a set of real-time attributes, and a set of private member functions.

```
CLASS <class name>, <description>
{
PRIVATE:
  CONFIGURATION ATTRIBUTES
    <type> <attribute name 1>, <description>
    <type> <attribute name 2>, <description>
              . . .
    <type> <attribute name N>, <description>

  REAL-TIME ATTRIBUTES
    <type> <attribute name 1>, <description>
    <type> <attribute name 2>, <description>
              . . .
    <type> <attribute name N>, <description>

  PRIVATE METHODS
    <type> <Method name 1>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>
    <type> <method name 2>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>
              . . .
    <type> <method name N>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>

PUBLIC:
  CONSTRUCTORS
    <constructor name 1>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>
    <constructor name 2>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>
          . . .
    <constructor name N>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>

  QUERY METHODS
    <type> <Method name 1>(<parameter-1, parameter-2, . . . , parameter-N),<desc.>
    <type> <method name 2>(<parameter-1, parameter-2, . . . , parameter-N),<descr.>
              . . .
   <type> <method name N>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>

  UPDATE METHODS
    <type> <method name 1>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>
    <type> <method name 2>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>
              . . .
    <type> <method name N>(<parameter-1, parameter-2, . . . , parameter-N), <desc.>

  ENCAPSULATED EVENTS
    <event name 1>, <description>
    <event name 2>, <description>
          . . .
    <event name N>, <description>

  EVENT HANDLERS
    <object type 1> <event name>
    <object type 2> <event name>
          . . .
    <object type N> <event name>

 ALARMS
    <alarm name 1>, <description>
    <alarm name 2>, <description>
          . . .
    <alarm name 3>, <description>

  HISTORY
    <history name 1>, <history type>, <description>
    <history name 2>, <history type>, <description>
          . . .
    <history name N>, <history type>, <description>
};
```

Figure 9.4 The REACT object interface

9.5 Summary

As noted earlier in the chapter, a complete discussion of these features is beyond the scope of this research. These features are discussed for completeness and because we make use of these features for our discussion control in Chapter 10. For most applications for which REACT is targeted, a history server, alarm server, and timekeeper would be required.

CHAPTER 10. IMPLEMENTING A CONTROL SYSTEM WITH REACT

The focus of this chapter is to show how the REACT database model can be used to implement one of the target applications: a control system. Human beings are not capable of monitoring and analyzing thousands of inputs in real-time and determining what systems are out of control or what outputs should be sent to keep a process under control or maintain conditions. Large real-time systems that need such large scale monitoring and control are now possible because of computers. Human beings should only be required to set general operating parameters, such as the flow rate in a pipeline, and can expect the control system to maintain these operating parameters. It is possible to create such a system using a REACT system. Special rule objects can be created using REACT to perform the control and monitoring functions that are beyond human capacity. The REACT encapsulated event/rule object interface greatly simplifies the task of specifying control and monitoring algorithms. With traditional systems, the events are based on database events. With REACT events correspond to real-world events such as "Pressure Changed", "Leak Detected", "Valve Closed", etc. With a REACT system, rule objects can also respond to multiple events, greatly simplifying the implementation of control and monitoring functions.

Implementing a control system requires cooperation between computer engineers, electrical engineers, and process engineers. Electrical engineers design the

signal conditioning and interfacing necessary to communicate real-world values (pressure, temperature, position, etc.) to a computer system and to send outputs that manipulate the state of the real-world. Process engineers supply algorithms that control some property in the real-world using these inputs and outputs. Computer engineers provide the hardware and software to form an environment in which the process engineer can easily implement his/her algorithms. The most desirable approach is to give the process engineer the tools necessary to directly specify control algorithms without intervention of the computer engineer. In the context of control, our goal with REACT has been to allow the process engineer to implement a complete control system by creating and instantiating objects in an active object-oriented database. The computer engineer in cooperation with the electrical engineer would create all of the interface object classes necessary to read process values from the real-world and to manipulate values in the real-world.

10.1 Basic Interface Objects for Input and Output

A REACT database used for control will need input/output interface objects to communicate with the real-world. In this section we will discuss four basic input/output types commonly used in control systems. The actual input/output driver subsystem required to communicate between interface objects and the real-world devices is beyond the scope of this research and will not be described.

10.1.1 Data types

There are two basic data types used for communicating with the real-world: analog and discrete. Analog data types are used for real-world entities that can be represented by a floating point value. Discrete data types are used for real-world entities that can be represented by a binary (true/false) value. There are other data types that might be used, but, for illustration purposes, only analog and discrete types will be considered.

10.1.2 Interface object types

Using analog and discrete data types, we will define four interface object types: analog input, discrete input, analog output, and discrete output. Analog input objects are used to communicate floating point values from the real-world to the database, discrete input objects are used to communicate discrete values from the real-world to the database, analog output objects are used to manipulate real-world values that can be represented by analog values, and discrete outputs are used to manipulate real world values that can be represented by a binary value. For example, the two input interface object types will have an encapsulated event for "Value Changed", and the two output interface object types will have "Send Output" member functions. The triggering of encapsulated events for discrete input interface objects and analog input interface objects is shown in Figure 10.1. These object types would typically have

```
discreteInput::newValue(boolean newValue, time_t scanTime)
{
 currentValue = newValue;
 lastScanTime = scanTime;
       .
       .   Determine state, alarming, history, etc.
       .
  triggerEvent(ValueChanged);
}

analogInput::newValue(float newValue, time_t scanTime)
{
 currentValue = newValue;
 lastScanTime = scanTime;
       .
       .   Unit conversion, alarming, history, etc.
       .
  triggerEvent(ValueChanged);
}
```

Figure 10.1 Interface member functions used to trigger encapsulated events

other encapsulated events, other member functions, alarms, history, and other features that could be enabled and disabled through configuration changes, but, for illustration purposes, we will only consider the basic features. Also, for illustration purposes, these are the only interface object types that will be considered even though some control systems might require other interface object types.

10.2 Rule Objects for Control and Monitoring

We will start by describing some rule object types that can be used to monitor and control large real-time systems. We have defined two types of object classes that would typically be used in control systems. There are other object types that could be

used for specific situations, but these two are the most common. We will also give examples of each object type later in this chapter.

10.2.1 Control objects

The first rule object type is a *control object*. A control object monitors real-world conditions by subscribing to "Value Changed" encapsulated events on interface objects, and maintains operating parameters such as temperature or flow by periodically sending outputs, through output interface objects, to manipulate the real-world. Control objects would be responsible for generating events or alarms, as appropriate, when they are unable to satisfactorily maintain operating parameters. We will give an example of a control object that implements the PID (Proportional-Integral-Derivative) control algorithm later in this chapter.

10.2.2 Monitor objects

Just as important as controlling a system, is the need for continuous fault monitoring. The second rule object type is a *monitoring object*. Monitoring rule objects monitor the state of the real world by subscribing to "Value Changed" events on input interface objects to detect conditions, possibly indicating failures or illegal operations, and then generate events or alarms as appropriate. Monitoring objects would never send outputs to control real-world parameters, but might possibly send outputs to shut down a real-world system when dangerous conditions exist. As an

example monitor object might detect real-world failures such as broken valves, leaks, over temperature, over speed, etc, and then generate events or alarms. We will give an example later in this chapter of an object that detects inoperable valves.

10.3 PID Control Example

We will now give an example using a REACT system for PID (Proportional-Integral-Derivative) control. This will require a rule object to perform the control algorithm and two analog interface objects, one analog input interface object to read the process variable, and one analog output interface object to set the manipulated variable. The rule object could then use these two interface objects as shown in Figure 10.2 to effect control of flow in a pipeline, for example. The flow object is an analog input interface object and has an encapsulated event "Value Changed" so that the control rule object is notified when new values arrive. In order to use this we must use a discrete approximation since we can not sample continuously or send outputs continuously. We will start with the text book positional algorithm:

$$MV(t)=K_P\left\{e(t) + \frac{1}{T_I}\int e(t)dt + T_D\frac{de(t)}{dt}\right\}$$

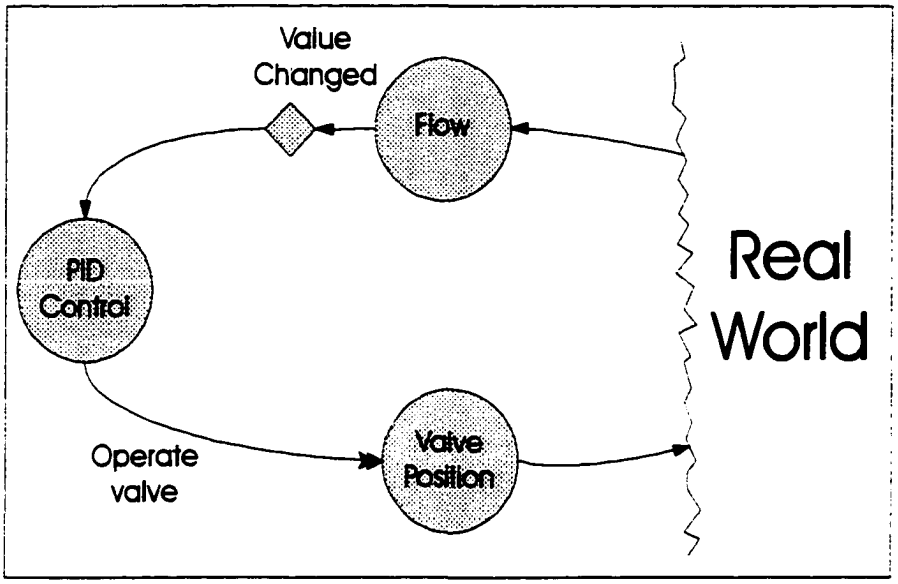

Figure 10.2 Diagram of object interaction for control

The integral is approximated using the trapezoid rule as follows:

$$I(t) \approx I(t-1) + \Delta t \frac{e(t)+e(t-1)}{2}$$

The expression dPV(t)/dt can be substituted for de/dt since they are equal when the set point does not change. Also, step changes in the set point will not cause an abnormally large derivative contribution. For simplicity, we will also use the straight line approximation for the derivative:

$$\frac{de(t)}{dt} \approx \frac{dPV(t)}{dt} \approx \frac{PV(t) - PV(t-1)}{\Delta t}$$

Making these two substitutions we have a discrete approximation for the positional algorithm which is suitable for implementation:

$$MV(t) \approx K_P\left\{e(t) + \frac{1}{T_I}\left(I(t-1)+\Delta t\frac{e(t)+e(t-1)}{2}\right) + T_D\frac{PV(t)-PV(t-1)}{\Delta t}\right\}$$

This becomes the rule that is implemented by the control rule object which is triggered by the encapsulated event "Value Changed". The event handler for this rule

```
EVENT[pv:ValueChanged](float newValue, TimeStamp time)
{
      float currentError, currentDerivative, currentMV;

      currentError = setpoint - newValue;
      deltaT = time - lastTime;
      totalIntegral +=
            deltaT * ((currentError + lastError) / 2.0);
      currentDerivative = (newValue - lastValue) / deltaT;
      currentMV = KP * ( currentError +
                        ((1.0/TI) *  totalIntegral) +
                        (TD * currentDerivative) );
      mv.sendOutput(currentMV);
      lastError = currentError;
      lastTime = time;
      lastValue = newValue;
}
```

Figure 10.3 Value changed event handler for PID control object

is given in Figure 10.3. This is for illustration purposes only. In a real system, we would require an event for "deviation exception" in the case that the control algorithm was unable to maintain the set point. Also there are better algorithms for the derivative that are not as noise-sensitive. This algorithm assumes that the process variable is sampled at regular intervals by the device driver so that the event "ValueChanged" will occur at regular intervals. If this were not the case we would have to use the time keeper to generate events at regular intervals to guarantee that this algorithm would execute at regular intervals. Once this rule object has been created, it can be used over and over again wherever a PID control algorithm is needed. This rule is applied to a specific situation by instantiating a rule object with parameters for the process variable object and manipulated variable object.

The REACT database model greatly simplifies the job of the process engineer. First, the REACT named encapsulated events have a one-to-one correspondence with real-world events. Second, the REACT rule object interface provides a simple environment in which to code control algorithms. To create the control class in this example, the control engineer only has to create a control class with one event handler for the "Value Changed" event and a simple constructor to initialize values. With an integrated development environment, this would amount to a few mouse clicks and then entering the code given in Figure 10.3. This approach also offers a clean separation of duties between the process engineer and the computer engineer. After the computer engineer creates all of the interface objects and the process engineer

creates the control class, it becomes trivial to apply the PID algorithm to new situations. In an integrated development environment, this could be as simple as a few mouse clicks.

Also, because of the REACT encapsulated event system, the flow readings can easily be used by other objects. These flow readings could be part of algorithms to detect pipeline leaks, and would be required for accounting purposes also.

10.4 Valve Monitoring Example

Our next example will be a monitoring rule object to detect faults in a valve as shown in Figure 10.4. We assume a valve that can be either opened or closed with two limit switches, one that indicates the valve is fully open, and the other that indicates the valve is fully closed. When the valve is opened or closed, it should be checked to make

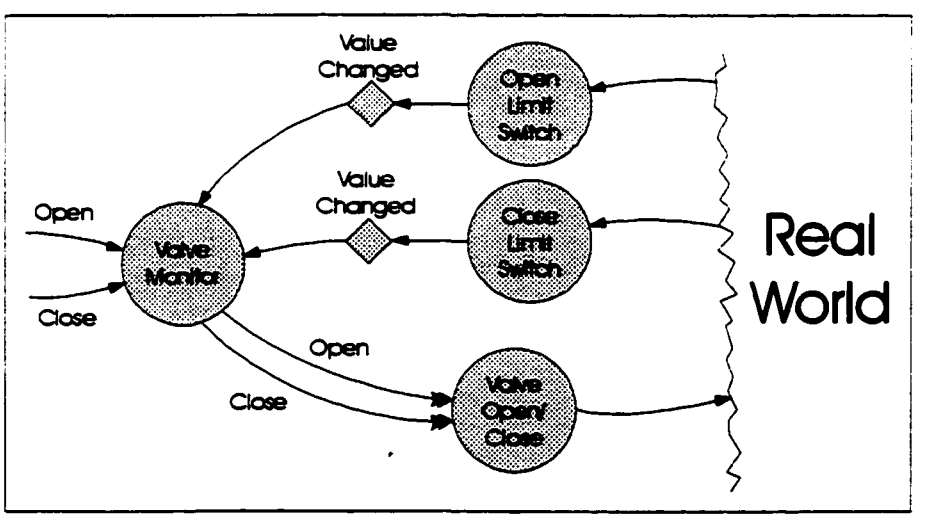

Figure 10.4 Valve monitor example

sure that the valve does in fact open or close. The valve does, however, take time to fully open or fully close after a command change so there must be a time delay before checking.

Also, the limit switches should not change when there is no command change. Previous control systems required the operator to monitor the valve operation manually, a tedious and error-prone activity. With a REACT system, the process is greatly simplified. We will create a rule object class to indirectly open and close the valve and check for error conditions. For simplicity the rule object will publish one event, "ValveError", which will be triggered if the object detects one of the errors given above. Also, open commands are ignored if the valve is not closed, and close commands are ignored if the valve is not open. There will be two member functions, one to open the valve, and one to close the valve as shown in Figure 10.5. Also, this object will subscribe to three events, one for "ValueChanged" of the open limit switch, one for "ValueChanged of the closed limit switch, and one for "OneShotExpired" of the time keeper, as shown in Figure 10.6. When the open member function is to be executed, the open member function on the output interface object is called, the state is changed to OPENING, and the one shot timer is started for 30 seconds. When the close member function is to be executed, the close member function on the output interface object is called, the state is changed to CLOSING, and the one shot timer is started for 30 seconds. If the open limit switch changes from TRUE to FALSE the state must be CLOSING. If the output limit switch changes from FALSE to TRUE the

```
void ValveMonitor::Open(void)
{
      if (state == CLOSED)
      {
            Valve.open();
            state = OPENING;
            StartOneShot(30);
      }
}

void ValveMonitor::Close(void)
{
      if (state == OPEN)
      {
            Valve.open();
            state = CLOSING;
            StartOneShot(30);
      }
}
```

Figure 10.5 Member functions for switch example

state must be OPENING. Any other state change of the output limit switch is an error. The logic for the close limit switch is similar.

10.5 Summary

We have given two examples of special rule objects that can be used as building blocks for a large control system. A large pipeline might have hundreds of open/close valves with limit switches or require hundreds of PID control loops. With a

```
EVENT[OpenLimitSwitch:ValueChanged](boolean newValue, TimeStamp time)
{
        if ((state == OPENING) && (newValue == TRUE))
        {
                state = OPEN;
                return;
        }
        else if ((state = CLOSING) && (newValue == FALSE))
        {
                return;
        }
        else
        {
                state = ERROR;
                TriggerEvent(ValveError);
        }
}



EVENT[CloseLimitSwitch:ValueChanged](boolean newValue, TimeStamp time)
{
        if ((state == CLOSING) && (newValue == TRUE))
        {
                state = CLOSED;
                return;
        }
        else if ((state = OPENING) && (newValue == FALSE))
        {
                return;
        }
        else
        {
                state = ERROR;
                TriggerEvent(ValveError);
        }

}



EVENT[TimeKeeper:OneShotExpired](TimeStamp time)
{
        if ((state == CLOSING)||(state = OPENING))
        {
                state = ERROR;
                TriggerEvent(ValveError);
                return;
        }
}
```

Figure 10.6 Event handlers for switch example.

REACT system, these rules are created once and then used over and over again. These control and monitoring functions are also very simple to implement with a REACT system by creating special rule objects. This type of functionality would be extremely difficult, if not impossible, to implement using a traditional rule base system.

CHAPTER 11. CONCLUSION AND FUTURE WORK

A powerful environment for the implementation of complex real-time systems has been developed. To bridge the gap between traditional databases and existing real-time systems we created the REACT database model. The model was designed to work with applications having very high transaction loads and moderate deadlines. The REACT database kernel was created to show that such a system could be built and could run on standard, commercially available hardware. Because of the importance of correctness for real-time systems we developed static analysis techniques such that we can guarantee the properties termination, confluence, and observable determinism will hold for a REACT databases.

We analyzed the performance of the REACT database assuming both single and multiple processors. For the multiple processor case, we developed a simulator based on the REACT database model, rule system, and concurrency control algorithms. The simulator enables us to evaluate the performance of REACT for various operating conditions. This is also a powerful tool for determining how REACT would perform for specific application requirements.

The REACT database provides a powerful environment for the implementation of control systems with a crisp division of labor between the electrical engineer,

computer engineer, and process engineer. This provides an environment in which reusable rule objects can be easily created for monitoring and control functions.

As for future work, there is also a need for very fast, very small databases that are located very close to the device or system being monitored. We plan to develop a version of the REACT database that will work in these environments. These systems would update objects with real-world values on the order of 100-1000 times per second, but would typically contain less than 50 objects, including interface objects. It would also be desirable to download objects from a host to perform control and or monitoring algorithms locally. As an example, some real-world systems will require control objects that operate at speed that is not possible with a host based control object.

We also plan to do research on a hierarchy of REACT databases, where the fastest and smallest databases exist on controllers mounted very close to a device or system that is being monitored and the slowest and largest database is located at a central location and contains a summary of data from all of the other databases.

APPENDIX A. STATIC ANALYSIS PSEUDO CODE

```
/*********************************************************************
*                          termination.txt                           *
*                                                                    *
*  Author: Don W. Carr                                               *
*                                                                    *
* Pseudo code for termination                                        *
*                                                                    *
*********************************************************************/

find_cycle(database D)
{
  remove all marks in D;
  for (all objects, O in D)
  {
    if (O is not marked for query)
    {
      L1 = empty list;
      L2 = search_for_query_cycle(L1, O);
      if (L2 != NULL)
      {
        print(L2);
        STOP;
      }
    }
    if (O is not marked for update)
    {
      L1 = empty list;
      L2 = search_for_update_cycle(L1, O);
      if (L2 != NULL)
      {
        print(L2);
        STOP;
      }
    }
  }
}
/*********************************************************************/

list search_for_query_cycle(list L1, dbobject O)
{
  mark O for query;
  if (O is in L1)
  {
    return L1
```

```
  }

  add O to L1;
  for (all objects, O2, on query list of O)
  {
    L2 = copy L1;
    L3 = search_for_query_cycle(L2, O2);
    if (L3 != NULL)
    {
      return L3;
    }
  }
}
/*****************************************************************/

list search_for_update_cycle(list L1, dbobject O)
{
  mark O for update;
  if (O is in L1)
  {
    return L1
  }

  add O to L1;
  for (all objects, O2, on update list of O)
  {
    L2 = copy L1;
    L3 = search_for_update_cycle(L2, O2);
    if (L3 != NULL)
    {
      return L3;
    }
  }

  for (all objects, O2, on event list of O)
  {
    L2 = copy L1;
    L3 = search_for_update_cycle(L2, O2);
    if (L3 != NULL)
    {
      return L3;
    }
  }
}

/*****************************************************************/
```

```
/*********************************************************************
*                          interfering.txt                           *
*                                                                    *
*  Author: Don W. Carr                                               *
*                                                                    *
* Pseudo code for interfering handlers.                              *
*                                                                    *
*********************************************************************/

interfering_handlers(database D)
{
  for (all objects, O in D)
  {
    if (has_interfering_handlers(O))
    {
      print O; STOP;
    }
  }
}

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

boolean has_interfering_handlers(dbobject O)
{
  for (all events, E on O)
  {
    N = number of subscribers to E;
    query = new list[N];
    update = new list[N];
    output_count = 0;
    i = 0;
    for (all subscribers, S to E)
    {
      query[i] = query_set(S, FALSE);
      change[i] = change_set(S);
      i++;
    }
    for (i = 0; i < n; i++)
    {
      for (int j = i + 1; j < n; j++)
      {
         if ((query[i] intersect change[j]) != empty set)
         {
           return TRUE;
         }
         if ((change[i] intersect query[j]) != empty set)
         {
           return TRUE;
```

```
          }
          if ((change[i] intersect change[j]) != empty set)
          {
            return TRUE;
          }
        }
      }
      return FALSE;
    }
    return FALSE;
}

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

list query_set(dbobject O, boolean query_this)
{
  L = empty list;
  for (all objects O2 on the query list of O)
  {
     add O2 to L;
     L = L union query_set(O2, TRUE);
  }
  if (!query_this)
  {
    for (all objects O2 on the update list of O)
    {
      L = L union query_set(O2, FASLE);
    }
    for (all objects O2 on the event list of O)
    {
      L = L union query_set(O2, FASLE);
    }
    return L;
  }
}

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

list change_set(dbobject O)
{
  L = empty list;
  for (all objects O2 on the update list of O)
  {
    add O to L;
    L = L union query_set(O2, FASLE);
  }
  for (all objects O2 on the event list of O)
  {
```

```
    add O to L;
    L = L union query_set(O2, FASLE);
  }
  return L;
}
/*****************************************************************/
```

```
/*********************************************************************
*                          conflicting.txt                           *
*                                                                    *
*  Author: Don W. Carr                                               *
*                                                                    *
* Static analysis pseudo code for conflicting handlers               *
*                                                                    *
*********************************************************************/

conflicting_handlers(database D)
{
  for (all objects, O in D)
  {
    if (has_conflicting_handlers(O))
    {
      print O;
      STOP;
    }
  }
}

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

boolean has_conflicting_handlers(dbobject O)
{
  for (all events, E on O)
  {
    output_count = 0;
    for (all subscribers, S to E)
    {
      if (sends_output(S))
      {
        output_count++;
      }
      if (output_count > 1)
      {
        return TRUE;
      }
    }
  }
  return FALSE;
}

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

boolean sends_output(dbobject O)
{
  if (O is an output interface object)
```

```
  {
     return TRUE;
  }

  for (all objects O2 on update list)
  {
    if (sends_output(O2))
    {
      return TRUE;
    }
  }

  for (all objects O2 on event list)
  {
    if (sends_output(O2))
    {
      return TRUE;
    }
  }

  return FALSE;
}

/*****************************************************************/
```

APPENDIX B. STATIC ANALYSIS SOURCE CODE

```
/*********************************************************************
*                              analysis.java                         *
*                                                                    *
*  Author: Don W. Carr                                               *
*                                                                    *
*  Source code for analysis class                                    *
*                                                                    *
*********************************************************************/


import java.awt.*;
import java.io.IOException;

public class analysis {

    public static void main(String args[]) {
        System.out.println("simple console application");
        System.out.println("");

        objdb db = new objdb();


        objlist l = db.find_cycle();
        if (l != null)
        {
            for (dbobject o = l.first(); o != null; o = l.next())
            {
                System.out.println(o.tag);
            }
        }
        else
        {
            System.out.println("No Cycles Found");
        }


        dbobject o;
        try
        {
            o = db.find_interfering_handlers();
        }
        catch(Exception e)
        {
```

```
            System.out.println("Exception: " + e.getMessage()
                      +" | " + e.toString());
            try
            {
                System.in.read();
            }
            catch (IOException ee)
            {
                return;
            }
            return;
        }
        if (o != null)
        {
            System.out.println("Event with interfering handlers: " +
                       o.tag + ":" + o.eventname);
        }
        else
        {
            System.out.println("No interfering handlers found");
        }

        o = db.find_conflicting_handlers();
        if (o != null)
        {
            System.out.println("Event with conflicting handlers: " +
                 o.tag + ":" + o.eventname);
        }
        else
        {
            System.out.println("No conflicting handlers found");
        }

        System.out.println("");
        System.out.println("(press Enter to exit)");
        try {
            System.in.read();
        } catch (IOException e) {
            return;
        }
    }
}
```

```
/*********************************************************************
*                            dbobject.java                           *
*                                                                    *
*  Author: Don W. Carr                                               *
*                                                                    *
*  Source code for dbobject class                                    *
*                                                                    *
*********************************************************************/

class dbobject
{
    public boolean sends_output()
    {
        if (this.output.get_num() > 0)
        {
            return true;
        }
        for (dbobject o = update.first(); o != null;
                        o = update.next())
        {
            if (o.sends_output())
            {
                return true;
            }
        }

        for (dbobject o = events.first(); o != null;
                o = events.next())
        {
            if (o.sends_output())
            {
                return true;
            }
        }
        return false;
    }
    public boolean conflicting_handlers()
    {
         int i = 0;
        for (int k=0; k < events.num_events; k++)
        {
          int n = events.events[k].subscribers.get_num();
          int output_count = 0;
          for (dbobject o = events.events[k].subscribers.first();
                 o != null; o = events.events[k].subscribers.next())
          {
            if (o.sends_output())
            {
```

```
                output_count++;
                if (output_count > 1)
                {
                    eventname = events.events[k].name;
                    return true;
                }
            }
        }

    }

    return false;
}
public objlist output;
public String eventname;
public event_list events;
public boolean interfering_handlers()
{
    int i = 0;
    for (int k=0; k < events.num_events; k++)
    {
      int n = events.events[k].subscribers.get_num();
      handler_sets hs[];
      hs = new handler_sets[n];
      for (i=0; i < n; i++)
      {
        hs[i] = new handler_sets();
      }
      //System.out.println("" +
        events.events[k].name + ": " +  n + " subscribers");
      i=0;
      for (dbobject o = events.events[k].subscribers.first();
           o != null; o = events.events[k].subscribers.next())
      {
        objlist l = o.query_set(false);
        try
        {
          //System.out.println("index = " + i + ", n = " + n);
          hs[i].query = l;
        }
        catch(Exception e)
        {
          System.out.println("Error: " + i + " " + e.toString());
          return false;
        }
        try
        {
          hs[i].change = o.change_set();
```

```
            hs[i].change.add(o);
          }
          catch(Exception e)
          {
            System.out.println("Error: " + i + e.toString());
            return false;
          }
          i++;
        }

        for (i = 0; i < n; i++)
        {
            for (int j = i + 1; j < n; j++)
            {
                if (hs[i].change.intersects(hs[j].change))
                {
                    eventname = events.events[k].name;
                    return true;
                }
                if (hs[i].change.intersects(hs[j].query))
                {
                    eventname = events.events[k].name;
                    return true;
                }
                if (hs[i].query.intersects(hs[j].change))
                {
                    eventname = events.events[k].name;
                    return true;
                }
            }
        }
    }

    return false;
}
public objlist change_set()
{
    objlist l = new objlist();
    for (dbobject o = update.first(); o != null;
                    o = update.next())
    {
        l.add(o);
        objlist l2 = o.change_set();
        l.union(l2);
    }

    for (dbobject o = events.first(); o != null;
                                o = events.next())
```

```
    {
        objlist l2 = o.change_set();
        l.union(l2);
    }
    System.out.print("Change set for " + this.tag + ": ");
    dbobject x = l.first();
    while (x != null)
    {
        System.out.print(" " + x.tag);
        x = l.next();
    }
    System.out.println("");

    return l;
}
public objlist query_set(boolean query_this)
{
    objlist l = new objlist();
    for (dbobject o = query.first(); o != null; o = query.next())
    {
        l.add(o);
        try
        {
          objlist l2 = o.query_set(true);
          l.union(l2);
        }
        catch(Exception e)
        {
          System.out.println("Error: " + e.toString());
          return l;
        }

    }
    if (!query_this)
    {
      for (dbobject o = update.first(); o != null;
                              o = update.next())
      {
          try
          {
            objlist l2 = o.query_set(false);
            l.union(l2);
          }
          catch(Exception e)
          {
            System.out.println("Error: " + e.toString());
            return l;
          }
```

```
            }
            for (dbobject o = events.first(); o != null;
                                        o = events.next())
            {
                try
                {
                  objlist l2 = o.query_set(false);
                  l.union(l2);
                }
                catch(Exception e)
                {
                  System.out.println("Error: " + e.toString());
                  return l;
                }

            }
        }
        System.out.print("Query set for " + this.tag + ": ");
        dbobject x = l.first();
        while (x != null)
        {
            System.out.print(" " + x.tag);
            x = l.next();
        }
        System.out.println("");
        return l;
    }
    public String tag;
    public dbobject(String t)
    {
        tag = t;
        query = new objlist();
        update = new objlist();
        output = new objlist();
        events = new event_list();
        mark = 0;

    }
    public objlist update;
    public objlist query;
    public int mark;
}
```

```
/*********************************************************************
*                              event_list.java                      *
*                                                                   *
*  Author: Don W. Carr                                              *
*                                                                   *
*  Source code for event_list class                                 *
*                                                                   *
*********************************************************************/

class event_list
    {
    private dbobject current_subscriber;
    public dbobject next()
    {
        if (current_event >= num_events)
        {
            return null;
        }
        while (current_subscriber == null)
        {
            current_event++;
            if (current_event >= num_events)
            {
                return null;
            }
            current_subscriber =
                    events[current_event].subscribers.first();
        }
        dbobject temp = current_subscriber;
        current_subscriber =
                    events[current_event].subscribers.next();
        return temp;
    }
    private int current_event;
    public dbobject first()
    {
        current_event = 0;
        if (num_events == 0)
        {
            return null;
        }
        current_subscriber = events[0].subscribers.first();
        return next();
    }
    public void add(objevent e)
    {
        try
        {
```

```
        events[num_events] = e;
        }
        catch(Exception ex)
        {
            System.out.println("Error in add: " + ex.toString());
            return;
        }
        num_events++;
    }
    public event_list()
    {
        events = new objevent[10];
        num_events = 0;
        current_event = 0;
        current_subscriber = null;
    }
    public int num_events;
    public objevent events[];
    }
```

```
/*************************************************************************
*                          handler_sets.java                             *
*                                                                        *
*  Author: Don W. Carr                                                   *
*                                                                        *
*  Source code for handler_sets class                                    *
*                                                                        *
*************************************************************************/

class handler_sets
    {
    public handler_sets()
    {
        query = new objlist();
        change = new objlist();
    }
    public objlist change;
    public objlist query;
    }
```

```
/*********************************************************************
*                              objdb.java                           *
*                                                                   *
*  Author: Don W. Carr                                              *
*                                                                   *
*  Source code for objdb class                                      *
*                                                                   *
*********************************************************************/

class objdb
    {
    public dbobject find_conflicting_handlers()
    {
        for (dbobject o = objs.first(); o != null; o = objs.next())
        {
            if (o.conflicting_handlers())
            {
                return o;
            }
        }
        return null;
    }
    public dbobject find_interfering_handlers()
    {
        for (dbobject o = objs.first(); o != null; o = objs.next())
        {
            if (o.interfering_handlers())
            {
                return o;
            }
        }
        return null;
    }
    public objdb()
    {
        objs = new objlist();
        dbobject oa, ob, oc, od, oe, of, og, oh;

        oa = new dbobject("A");
        objs.add(oa);

        ob = new dbobject("B");
        objs.add(ob);

        oc = new dbobject("C");
        objs.add(oc);

        od = new dbobject("D");
```

```
        objs.add(od);

        oe = new dbobject("E");
        objs.add(oe);

        of = new dbobject("F");
        objs.add(of);

        og = new dbobject("G");
        objs.add(og);

        oh = new dbobject("H");
        objs.add(oh);

        /******************/
        objevent e;
        e = new objevent("E1");
        try
        {
        e.subscribers.add(ob);
        }
        catch(Exception ex)
        {
            System.out.println("Exception: " + ex.toString());
            return;
        }
        oa.events.add(e);
        ob.query.add(oh);

        e = new objevent("E1");
        e.subscribers.add(od);
        e.subscribers.add(oe);
        oc.events.add(e);
        oe.update.add(of);
        od.output.add(og);
        of.query.add(od);
        of.output.add(oh);
        /******************/
    }
    public void mark_all(int m)
    {
        for (dbobject o = objs.first(); o != null; o = objs.next())
        {
            o.mark = m;
        }
    }
    public objlist search_for_cycle(int mark, objlist l,
                                    dbobject o, boolean isquery)
```

```
{
    if ((o.mark == mark) && (l.in(o)))
    {
        l.delete_before(o);
        return l;
    }
    else
    {
        o.mark = mark;
        l.add(o);
        if (!isquery)
        {
          for (dbobject o2 = o.update.first();
                              o2 != null; o2 = o.update.next())
          {
              objlist l2 = new objlist(l);
              objlist l3 = search_for_cycle(mark, l2, o2, false);
              if (l3 != null)
              {
                  return l3;
              }
          }
          for (dbobject o2 = o.events.first();
                              o2 != null; o2 = o.events.next())
          {
              objlist l2 = new objlist(l);
              objlist l3 = search_for_cycle(mark, l2, o2, false);
              if (l3 != null)
              {
                  return l3;
              }
          }
        }
        else
        {
          for (dbobject o2 = o.query.first();
                          o2 != null; o2 = o.query.next())
          {
              objlist l2 = new objlist(l);
              objlist l3 = search_for_cycle(mark, l2, o2, true);
              if (l3 != null)
              {
                  return l3;
              }
          }
        }

    }
```

```
    return null;
}
public objlist find_cycle()
{
    int current_mark = 1;
    mark_all(0);
    for (dbobject o = objs.first(); o != null; o = objs.next())
    {
        if (o.mark != 0)
        {
            continue;
        }
        objlist l = new objlist();
        objlist l2 = search_for_cycle(current_mark, l, o, false);
        if (l2 != null)
        {
            return l2;
        }
        current_mark++;
        l = new objlist();
        l2 = search_for_cycle(current_mark, l, o, true);
        if (l2 != null)
        {
            return l2;
        }
        current_mark++;
    }
    return null;
}
public objlist objs;
}
```

```
/*********************************************************************
*                              objevent.java                        *
*                                                                   *
*  Author: Don W. Carr                                              *
*                                                                   *
*  Source code for objevent class                                   *
*                                                                   *
*********************************************************************/

class objevent
    {
    public objevent(String n)
    {
        name = n;
        subscribers = new objlist();
    }
    public objevent()
    {
        name = "None";
        subscribers = new objlist();
    }
    public objlist subscribers;
    public String name;
    }
```

```
/*********************************************************************
*                              objlist.java                          *
*                                                                    *
*  Author: Don W. Carr                                               *
*                                                                    *
*  Source code for objlist class                                     *
*                                                                    *
*********************************************************************/

class objlist
    {
    public int get_num()
    {
        return num_objs;
    }
    public boolean intersects(objlist l)
    {
        for (dbobject o = l.first(); o != null; o = l.next())
        {
            if (in(o))
            {
                return true;
            }
        }
        return false;

    }
    public void union(objlist l)
    {
        for (dbobject o = l.first(); o != null; o = l.next())
        {
            if (!this.in(o))
            {
                this.add(o);
            }
        }
    }
    public void delete_before(dbobject o)
    {
        int obj_pos = -1;
        for (int i = 0; i < num_objs; i++)
        {
            if (objs[i] == o)
            {
                obj_pos = i;
                break;
            }
        }
```

```
    if (obj_pos > 0)
    {
        num_objs -= obj_pos;
        for (int i = 0; i < num_objs; i++)
        {
            objs[i] = objs[i + obj_pos];
        }
    }
}
public dbobject next()
{
    if (current >= num_objs)
    {
        return null;
    }
    else
    {
        current++;
        return objs[current-1];
    }
}
private int current;
public dbobject first()
{
    current = 0;
    return next();
}
public boolean in(dbobject o)
{
    for (int i = 0; i < num_objs; i++)
    {
        if (objs[i] == o)
        {
            return true;
        }
    }
    return false;
}
public objlist()
{
    objs = new dbobject[100];
    num_objs = 0;
}
public objlist(objlist l)
{
    objs = new dbobject[100];
    num_objs = l.num_objs;
    for (int i=0; i < num_objs; i++)
```

```
    {
        objs[i] = l.objs[i];
    }
}
public void add(dbobject o)
{
    objs[num_objs] = o;
    num_objs++;
}
private int num_objs;
private dbobject objs[];
}
```

APPENDIX C. REACT KERNEL SOURCE CODE

```
/*****************************************************************
*                          OODB.CPP                              *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
* Contains source code for object database class                 *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <string.h>

#include "oodb.h"

object_database_t *oodb;

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

event_id_t db_object_t::subscribe_event(char *key, char *event_name)
{
  return oodb->subscribe_event(oid, key, event_name);
}

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

void db_object_t::trigger_event(short event_number)
{
  oodb->trigger_event(oid, event_number);
}

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

object_database_t::object_database_t(long max_objects)
{
  next_eid = 1;
  num_pending_events = 0;
  obj_list = new obj_list_t(max_objects);
}

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

void object_database_t::insert_object(db_object_t *obj)
{
  obj_list->insert(obj);
}
```

```
/*****************************************************************

db_object_t *object_database_t::get_object(char *key)
{
  return obj_list->get_object(key);
}

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

event_id_t object_database_t::subscribe_event(object_id_t subscriber,
                      char *key,
                      char *event)
{
  db_object_t *obj;
  event_list_t *elist;
  char **events;
  int n_events;

  obj_list->get(key, &obj, &elist);

  if ((obj == NULL) || (elist == NULL))
  {
    return 0;
  }

  short event_number = -1;
  n_events = obj->published_events(&events);

  for (int i=0; i < n_events; i++)
  {
    if (0 == strcmpi(events[i], event))
    {
      event_number = i;
      break;
    }
  }

  if (event_number == -1)
  {
    return 0;
  }

  event_id_t eid = next_event_id();
  elist->add_subscriber(event_number, subscriber, eid);
  return eid;
}

/*****************************************************************/
```

```
void object_database_t::trigger_event(object_id_t event_originator,
                short event_number)
{
  pending_events[num_pending_events].event_originator =
event_originator;
  pending_events[num_pending_events].event_number = event_number;
  num_pending_events++;
}

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

void object_database_t::trigger_pending_events(void)
{
  for (int i=0; i < num_pending_events; i++)
  {
    db_object_t *obj;
    event_list_t *elist;

    obj_list->get(pending_events[i].event_originator, &obj, &elist);

    if (elist == NULL)
    {
      continue;
    }

    event_list_elem_t *elem;

    for (elem = elist->list[pending_events[i].event_number];
       elem != NULL;
       elem = elem->next)
    {
      db_object_t *obj;
      event_list_t *elist;
      obj_list->get(elem->subscriber, &obj, &elist);
      if (obj == NULL)
      {
      continue;
      }
      obj->event_occurred(elem->eid);
    }
  }
  num_pending_events = 0;
}


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

int object_database_t::execute_transaction(char *obj_key,
```

```
                                    char *method,
                                    int n_args,
                                    arg_t *args)
{
  db_object_t *obj;
  event_list_t *elist;

  obj_list->get(obj_key, &obj, &elist);
  if (obj == NULL)
  {
    return -1;
  }

  obj->execute_method(method, n_args, args);

  return 0;

}

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

db_object_t *object_database_t::get_object(char *key)
{
  db_object_t *obj;
  event_list_t *list;
  obj_list->get(key, &obj, &list);
  return obj;
}

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

event_id_t object_database_t::next_event_id(void)
{

  next_eid++;
  return next_eid;
}

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

agent_t *get_agent_object(char *key)
{
  db_object_t *obj;
  obj = oodb->get_object(key);
  if (obj == NULL)
  {
    return NULL;
  }
```

```
  if (obj->object_type() != AGENT_OBJECT)
  {
    return NULL;
  }

  return (agent_t *) obj;
}

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

input_t *get_input_object(char *key)
{
  db_object_t *obj;
  obj = oodb->get_object(key);
  if (obj == NULL)
  {
    return NULL;
  }
  if (obj->object_type() != INPUT_OBJECT)
  {
    return NULL;
  }
  return (input_t *) obj;
}
```

```
/*****************************************************************
*                          OBJLIST.CPP                           *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*   Contains source code for creating and accessing object       *
*   hash table                                                   *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <math.h>
#include <malloc.h>
#include <string.h>

#include "oodb.h"


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

static int is_prime(long n)
{

  if ((n % 2) == 0)
  {
    return 0;
  }

  long max = (long) sqrt(n);

  for (long d = 3; d <= max; d += 2)
  {
    if ((n % d) == 0)
    {
      return 0;
    }
  }
  return 1;
}

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

static long next_prime(long n)
{
  if ((n % 2) == 0) // Start with an odd number.
  {
    n++;
  }
```

```
  while (!is_prime(n))
  {
    n += 2;
  }
  return n;
}

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

static long hashpjw(char *s, long max)
{
  char *p;
  unsigned long g;
  unsigned long h = 0;

  for (p=s; *p != '\0'; p++)
  {
    h = (h << 4) + (*p);
    g = h & 0xF0000000;
    if (g != 0)
    {
      h = h ^ (g >> 24);
      h = h ^ g;
    }
  }
  return h % max;
}

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


obj_list_t::obj_list_t(long max_objects)
{
  max = next_prime(max_objects);
  long size = max * sizeof(obj_list_element_t);
  list = (obj_list_element_t *) malloc(size);
  MALLOC_CHECK(list);
  for (long i=0; i < max; i++)
  {
    list[i].obj = NULL;
    list[i].elist = NULL;
  }
}

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

long n_skip = 0;
```

```
void obj_list_t::insert(db_object_t *obj)
{
  long hash_value = hashpjw(obj->key, max);
  int skip = 0;
  while (list[hash_value].obj != NULL)
  {
    skip = 1;
    hash_value = (hash_value + 1) % max;
  }
  if (skip)
  {
    n_skip++;
  }
  obj->oid = hash_value;
  char **events;
  int num_events;

  num_events = obj->published_events(&events);
  list[hash_value].elist = new event_list_t(num_events);
  list[hash_value].obj = obj;
}

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

void obj_list_t::get(char *key, db_object_t **obj, event_list_t
**elist)
{
  long hash_value = hashpjw(key, max);
  while (list[hash_value].obj != NULL)
  {
    if (0 == strcmpi(key, list[hash_value].obj->key))
    {
      *obj = list[hash_value].obj;
      *elist = list[hash_value].elist;
      return;
    }
    hash_value = (hash_value + 1) % max;
  }
  return;
}

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

void obj_list_t::get(object_id_t oid,
                     db_object_t **obj,
                     event_list_t **elist)
{
  *obj = list[oid].obj;
```

```
  *elist = list[oid].elist;
  return;
}

/*****************************************************************/
```

```
/*****************************************************************
*                          ELIST.CPP                             *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*  Code for event list class                                     *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <malloc.h>

#include "oodb.h"


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

event_list_t::event_list_t(int n_events)
{
  n = n_events;
  list = (event_list_elem_t **) malloc(sizeof(event_list_elem_t *));
  MALLOC_CHECK(list);
  for (short i=0; i < n; i++)
  {
    list[i] = NULL;
  }
}

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

void event_list_t::add_subscriber(short event_number,
                          object_id_t subscriber,
                          event_id_t eid)
{
  event_list_elem_t *new_elem;

  new_elem = new event_list_elem_t;

  new_elem->subscriber = subscriber;
  new_elem->eid = eid;

  new_elem->next = list[event_number];
  list[event_number] = new_elem;
}
/*****************************************************************/
```

```
/*****************************************************************
*                          OODBMAIN.CPP                          *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*   Contains main program to load test database and run.         *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <time.h>
#include <new.h>

#include "oodb.h"

extern object_database_t *oodb;


#define MAX_INPUT 100


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

void main(int argc, char *argv[])
{
  char *obj_key;
  char *method;
  arg_t *args;
  int n_args;
  long i;
  char input_key[20], rule_key[20], agent_key[20];
  time_t start_time, end_time;

  set_new_handler(common_new_handler);

  oodb = new object_database_t(1000);

  for (i=0; i < MAX_INPUT; i++)
  {
    sprintf(input_key, "in%ld", i);
    sprintf(rule_key, "rule%ld", i);
    sprintf(agent_key, "agent%ld", i);

    input_t *inp;
    inp = new input_t(input_key);
    oodb->insert_object(inp);
    inp->startup_phase_1();
    inp->startup_phase_2();
```

```
    agent_t *agent;
    agent = new agent_t(agent_key);
    oodb->insert_object(agent);
    agent->startup_phase_1();
    agent->startup_phase_2();


    rule_t *rule;
    rule = new rule_t(rule_key, input_key, agent_key);
    oodb->insert_object(rule);
    rule->startup_phase_1();
    rule->startup_phase_2();
  }


  arg_t arg[1];
  arg[0].type = float_type;
  arg[0].value.float_val = 4.0;

  time(&start_time);
  for (i=0; (i < 1000000); i++)
  {
    sprintf(input_key, "in%ld", i % MAX_INPUT);
    oodb->execute_transaction(input_key, "update", 1, arg);
    oodb->trigger_pending_events();
  }

  time(&end_time);
  printf("Elapsed time = %ld\n", long(end_time - start_time));

  arg[0].type = long_type;
  oodb->execute_transaction("rule1", "GetCount", 1, arg);
  printf("Event Count = %ld\n", arg[0].value.long_val);

}
```

```
/*****************************************************************
*                          INPUT.CPP                             *
*                                                                *
*   Author: Don W. Carr                                          *
*                                                                *
*  Source code for example input object                          *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <string.h>

#include "oodb.h"


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

input_t::input_t(char *nkey)
{
  strcpy(key, nkey);
  value = 0;
}

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

void input_t::update(float new_value)
{
  value = new_value;
  trigger_event(0);
}

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

float input_t::get_value(void)
{
  return value;
}

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

int input_t::published_events(char ***names)
{
  static char *s[] =
  {
    "ValueChanged"
  };
  *names = s;
  return 1;
```

```
}

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

int input_t::methods_with_update(char ***signatures)
{
  static char *s[] =
  {
    "void update(in float new_value);"
  };
  *signatures = s;
  return 1;
}

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

int input_t::methods_without_update(char ***signatures)
{
  static char *s[] =
  {
    "void get_value(out float current_value);"
  };
  *signatures = s;
  return 1;
}

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

int input_t::execute_method(char *method, int n_args, arg_t *args)
{
  if (0 == strcmpi(method, "update"))
  {
    if (n_args != 1)
    {
      return -1;
    }
    if (args[0].type != float_type)
    {
      return -1;
    }
    update(args[0].value.float_val);
    return 0;
  }
  if (0 == strcmpi(method, "GetValue"))
  {
    if (n_args != 1)
    {
      return -1;
```

```
    }
    if (args[0].type != float_type)
    {
      return -1;
    }
    args[0].value.float_val = value;
    return 0;
  }
  return -1;
}

/*****************************************************************/
```

```
/*********************************************************************
*                          RULE.CPP                                  *
*                                                                    *
* Author: Don W. Carr                                                *
*                                                                    *
*  Source code for example rule object.                              *
*                                                                    *
*********************************************************************/

#include <stdio.h>
#include <string.h>

#include "oodb.h"


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

rule_t::rule_t(char *nkey, char *skey, char *akey)
{
  strcpy(key, nkey);
  strcpy(subject_key, skey);
  strcpy(agent_key, akey);
  change_eid = 0;
  subject_object = NULL;
  agent_object = NULL;
  event_count = 0;
}

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

void rule_t::startup_phase_2(void)
{
  change_eid = subscribe_event(subject_key, "ValueChanged");
  agent_object = get_agent_object(agent_key);
  subject_object = get_input_object(subject_key);
}

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

void rule_t::get_count(int *value)
{
  *value = event_count;
  return;
}

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

int rule_t::published_events(char ***names)
```

```
{
  static char *s[] =
  {
    "ValueChanged"
  };
  *names = s;
  return 1;
}

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

int rule_t::methods_with_update(char ***signatures)
{
  *signatures = NULL;
  return 0;
}

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

int rule_t::methods_without_update(char ***signatures)
{
  static char *s[] =
  {
    "void get_count(out int current_count);"
  };
  *signatures = s;
  return 1;
}

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

int rule_t::execute_method(char *method, int n_args, arg_t *args)
{
  if (0 == strcmpi(method, "GetCount"))
  {
    if (n_args != 1)
    {
      return -1;
    }
    if (args[0].type != long_type)
    {
      return -1;
    }
    args[0].value.long_val = event_count;
    return 0;
  }
  return -1;
}
```

```
/*****************************************************************/

void rule_t::event_occurred(event_id_t eid)
{
  if (eid == change_eid)
  {
    event_count++;
    if (agent_object != NULL)
    {
      agent_object->send(3.0);
    }
  }
}

/*****************************************************************/
```

```
/*****************************************************************
*                          AGENT.CPP                             *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*  Source code for example agent object                          *
*                                                                *
*****************************************************************/
#include <stdio.h>
#include <string.h>

#include "oodb.h"

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

agent_t::agent_t(char *nkey)
{
  strcpy(key, nkey);
  last_output = 0;
}

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

void agent_t::send(float output_value)
{
  last_output = output_value;
  trigger_event(0);
}

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

void agent_t::get_last_output(float *value)
{
  *value = last_output;
  return;
}

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

int agent_t::published_events(char ***names)
{
  static char *s[] =
  {
    "OutputSent"
  };
  *names = s;
  return 1;
}
```

```
/*****************************************************************/

int agent_t::methods_with_update(char ***signatures)
{
  static char *s[] =
  {
    "void send(in float new_value);"
  };
  *signatures = s;
  return 1;
}

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

int agent_t::methods_without_update(char ***signatures)
{
  static char *s[] =
  {
    "void get_last_output(out float last_out);"
  };
  *signatures = s;
  return 1;
}

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

int agent_t::execute_method(char *method, int n_args, arg_t *args)
{
  if (0 == strcmpi(method, "send"))
  {
    if (n_args != 1)
    {
      return -1;
    }
    if (args[0].type != float_type)
    {
      return -1;
    }
    send(args[0].value.float_val);
    return 0;
  }
  if (0 == strcmpi(method, "GetLastOutput"))
  {
    if (n_args != 1)
    {
      return -1;
    }
    if (args[0].type != float_type)
```

```
    {
      return -1;
    }
    args[0].value.float_val = last_output;
    return 0;
  }
  return -1;
}

/*****************************************************************/
```

```
/*********************************************************************
*                          OODB.H                                    *
*                                                                    *
* Author: Don W. Carr                                                *
*                                                                    *
*   Header file for database kernel                                  *
*                                                                    *
*********************************************************************/


#ifndef __OODB_INC__
#define __OODB_INC__

#include "stdio.h"
#include "common.h"

enum object_type_t
{
  ANALOG_INPUT,
  ANALOG_OUTPUT,
  DISCRETE_INPUT,
  DISCRETE_OUTPUT,
  VAR_OBJECT,
  CALC_OBJECT,
  DCALC_OBJECT,
  DVAR_OBJECT,
  CONTROL_OBJECT,
  AGENT_OBJECT,
  RULE_OBJECT,
  INPUT_OBJECT,
};

enum pv_type_t
{
  ANALOG_VALUE,
  DISCRETE_VALUE,
  UNDEFINED_VALUE,
  INTEGER_VALUE,
};

enum object_state_t
{
  LOW_ALARM = 0,
  LOW_CAUTION = 1,
  NORMAL = 2,
  HIGH_CAUTION = 3,
  HIGH_ALARM = 4,
  ALARM = 5,
```

```
  POINT_FAILED = 6,
  DEV_ALARM,
  DEV_CAUTION,
  CAUTION,
};

typedef long event_id_t;
typedef long object_id_t;
typedef long reference_t;
typedef long analog_expr_t;
typedef long discrete_expr_t;

union value_t
{
  float float_val;
  int discrete_val;
  int int_val;
  long long_val;
  char char_val;
  char *string_val;
};

enum type_t
{
  float_type,
  int_type,
  long_type,
  discrete_type,
  char_type,
  string_type
};

struct arg_t
{
  type_t type;
  value_t value;
};

class db_object_t
{
public:
  object_id_t oid;
  char key[21];
  char description[31];
  object_state_t object_state;
  object_state_t last_object_state;
  long execution_group;
```

```
  virtual void startup_phase_1(void) {return;};
  virtual void startup_phase_2(void) {return;};

  event_id_t subscribe_event(char *object_key, char *event);
  virtual void event_occurred(event_id_t eid) {return;};
  void trigger_event(short event_number);

  virtual int published_events(char ***names) = 0;
  virtual int methods_with_update(char ***signatures) = 0;
  virtual int methods_without_update(char ***signatures) = 0;
  virtual int execute_method(char *method, int n_args, arg_t *args) =
0;
  virtual object_type_t object_type(void) = 0;
};

class input_t : public db_object_t
{
private:
  float value;
public:
  input_t(char *key);
  void update(float new_value);
  float get_value(void);
  void event_occurred(event_id_t eid) {return;};
  int published_events(char ***names);
  int methods_with_update(char ***signatures);
  int methods_without_update(char ***signatures);
  int execute_method(char *method, int n_args, arg_t *args);
  object_type_t object_type(void) {return INPUT_OBJECT;};
};

class agent_t : public db_object_t
{
private:
  float last_output;
public:
  agent_t(char *nkey);
  void send(float output_value);
  void get_last_output(float *current_count);
  int published_events(char ***names);
  int methods_with_update(char ***signatures);
  int methods_without_update(char ***signatures);
  int execute_method(char *method, int n_args, arg_t *args);
  object_type_t object_type(void) {return AGENT_OBJECT;};
};

class rule_t : public db_object_t
{
```

```
private:
  char subject_key[21];
  char agent_key[21];
  input_t *subject_object;
  agent_t *agent_object;
  long event_count;
  event_id_t change_eid;
public:
  rule_t(char *nkey, char *skey, char *akey);
  void get_count(int *current_count);
  void startup_phase_2(void);
  void event_occurred(event_id_t eid);
  int published_events(char ***names);
  int methods_with_update(char ***signatures);
  int methods_without_update(char ***signatures);
  int execute_method(char *method, int n_args, arg_t *args);
  object_type_t object_type(void) {return RULE_OBJECT;};
};

class analog_object_t : public db_object_t
{
public:
  float pv;
  char eu[8];
  int decimal;
  analog_object_t(void);
  inline float get_pv(void) {return pv;};
  pv_type_t pv_type(void) {return ANALOG_VALUE;};
};

class analog_alarm_object_t : public analog_object_t
{
  float hi_alarm;
  float hi_caution;
  float lo_caution;
  float lo_alarm;
  float deadband;
  logical hi_alarm_enable;
  logical lo_alarm_enable;
  logical hi_caution_enable;
  logical lo_caution_enable;
  logical hi_alarm_shutdown;
  logical lo_alarm_shutdown;
  void check_alarms(void);
};

struct ai_realtime_t
{
```

```
  int update_time_stamp;
  float raw_value;
};

class ai_object_t : public analog_alarm_object_t
{
public:
  ai_realtime_t rt;
  float pv_last;
  int update_time_stamp;
  int last_update_time_stamp;
  float conversion_factor;
  float eu_hi;
  float eu_lo;
  float raw_lo;
  float raw_hi;
  float zero_cutoff;
  void check_alarms(void);
  void set_conversion(float rl, float rh, float eul, float euh);
  void set_conversion(void);
public:
  ai_object_t(void);
  void update(float new_raw_value);
  object_type_t object_type(void) {return ANALOG_INPUT;};
};

class ao_object_t : public analog_object_t
{
public:
  int output_time_stamp;
  void send(float value);
  float output_limit_hi;
  float output_limit_lo;
  float eu_hi;
  float eu_lo;
  float raw_lo;
  float raw_hi;
public:
  object_type_t object_type(void) {return ANALOG_OUTPUT;};
};

class control_object_t : public analog_object_t
{
private:
  float setpoint_limit_hi;
  float setpoint_limit_lo;
  float deviation_alarm;
  float deviation_caution;
```

```
  float deviation_delay;
  double dev_caut_detect_time;
  double dev_alm_detect_time;
  logical dev_alarm_enable;
  logical dev_alarm_shutdown;
  logical dev_caution_enable;
  float p_gain;
  float i_time;
  float d_time;
  float setpoint;
  float last_input;
  float last_output;
  float last_error;
  float integrated_error;
  double last_time;
  logical control_enabled;
  logical ramp_is_on;
  float ramp_increment;
  float ramp_value;
  int ramp_counter;
  int updates_per_evaluation;
  int delay_counter;
  ao_object_t *ao_point;
  ai_object_t *ai_point;
public:
  void change_setpoint(float val, float ramp_time);
  void start_control(void);
  void stop_control(void);
  object_type_t object_type(void) {return CONTROL_OBJECT;};
};

class discrete_object_t : public db_object_t
{
public:
  int pv;
  char *pv_string;
  char hi_desc[12];
  char lo_desc[12];
  void display(void);
  void display_pv(void);
  inline int get_pv(void) {return pv;};
  pv_type_t pv_type(void) {return DISCRETE_VALUE;};
};


class di_object_t : public discrete_object_t
{
public:
```

```
  int pv_last;
  int update_time_stamp;
  int last_update_time_stamp;
  int alarm_state;
  void check_alarms(void);
public:
  void update(int new_value);
  object_type_t object_type(void) {return DISCRETE_INPUT;};
};

class do_object_t : public discrete_object_t
{
public:
  int output_time_stamp;
  void send(int value);
  object_type_t object_type(void) {return DISCRETE_OUTPUT;};
};

class calc_object_t : public analog_object_t
{
public:
  analog_expr_t expression;
  char *expr_string;
  void evaluate(void);
  void parse_expr(void);
  object_type_t object_type(void) {return CALC_OBJECT;};
};

class dcalc_object_t : public discrete_object_t
{
public:
  discrete_expr_t expression;
  char *expr_string;
  void evaluate(void);
  void parse_expr(void);
  object_type_t object_type(void) {return DCALC_OBJECT;};
};

class var_object_t : public analog_object_t
{
public:
  inline void set_pv(float new_pv) {pv = new_pv;};
  object_type_t object_type(void) {return VAR_OBJECT;};
};

class dvar_object_t : public discrete_object_t
{
public:
```

```
  logical value_stored;
  void set_pv(int new_pv);
  object_type_t object_type(void) {return DVAR_OBJECT;};
};

struct event_list_elem_t
{
  object_id_t subscriber;
  event_id_t eid;
  struct event_list_elem_t *next;
};

class event_list_t
{
public:
  event_list_elem_t **list;
  short n;
  event_list_t(int n_events);
  void add_subscriber(short event_number,
            object_id_t subscriber,
            event_id_t eid);
};

struct obj_list_element_t
{
  db_object_t *obj;
  event_list_t *elist;
};

class obj_list_t
{
private:
  long max;
  obj_list_element_t *list;
public:
  obj_list_t(long max_objects);
  void insert(db_object_t *obj);
  void get(char *key, db_object_t **obj, event_list_t **elist);
  void get(object_id_t oid, db_object_t **obj, event_list_t **elist);
};

struct pending_event_t
{
  object_id_t event_originator;
  short event_number;
};

class object_database_t
```

```
{
private:
  obj_list_t *obj_list;
  event_id_t next_eid;
  pending_event_t pending_events[100];
  short num_pending_events;
public:
  object_database_t(long max_objects);
  void insert_object(db_object_t *obj);
  event_id_t subscribe_event(object_id_t subscriber, char *key, char
*event);
  void trigger_event(object_id_t event_originator, short n);
  void trigger_pending_events(void);
  db_object_t *get_object(char *key);
  event_id_t next_event_id(void);
  int execute_transaction(char *object_key,
                    char *function,
                    int n_args,
                    arg_t *args);
  db_object_t *get_object(char *key);
};

agent_t *get_agent_object(char *key);
input_t *get_input_object(char *key);

#endif
```

APPENDIX D. SIMULATOR SOURCE CODE

```
/*****************************************************************
*                          SIM.CPP                               *
*                                                                *
*  Author: Don W. Carr                                           *
*                                                                *
*  This file contains the procedures to read the simulation      *
* input and run the simulation.                                  *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "compat.h"
#include "sim.h"
#include "trans.h"
#include "util.h"
#include "random.h"

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

running_t run;
waiting_t wait;
FILE *fp;

void simulate(input_data_t *in, int *overflow)
{

  memset(&run, '\0', sizeof(run));
  memset(&wait, '\0', sizeof(wait));
  run.reset();
  wait.reset();
  *overflow = 0;
  current_time = 0;
  int next_event;
  transaction_t t;
  double total_transaction_execution_time = 0;
  double total_transaction_time_in_system = 0;
  double last_arrive_time = 0;
  double next_arrive_time =
          last_arrive_time + get_next_arrival_time(1.0 /
```

```
in->arrival_rate);
  int done = 0;
  first_seeds();
  for (long i=0; (i < 10000) && !done;)
  {
    if ((i%1000) == 0)
    {
      next_seeds();
    }
    if (run.n_running() > 0)
    {
      double finish_time = run.next_finish_time();
      if (finish_time < next_arrive_time)
      {
      next_event = TRANS_COMPLETE;
      }
      else
      {
      next_event = TRANS_ARRIVE;
      }
    }
    else
    {
      next_event = TRANS_ARRIVE;
    }
    switch (next_event)
    {
      case TRANS_ARRIVE:

      if (next_arrive_time > current_time)
      {
        current_time = next_arrive_time;
      }
      double delta_t = get_next_arrival_time(1.0 / in->arrival_rate);
      t.arrive_time = next_arrive_time;
      last_arrive_time = next_arrive_time;
      next_arrive_time = last_arrive_time + delta_t;
      t.num_functions = get_n_functions(in->n_functions);
      t.execution_time = get_execution_time(t.num_functions,
                                            in->execution_time);
      t.deadline = current_time + in->deadline;
      t.group = get_next_group();
      t.num_waiting = 0;
      t.num_waiting += run.incompatible_with(t.group,
in->p_incompat);
      int num_compare = 0;
      num_compare += run.num_compare(t.num_functions);
      num_compare += wait.num_compare(t.num_functions);
```

```
      double check_delay = num_compare * in->overhead;
      current_time += check_delay;
      wait.insert_transaction(&t, in->p_incompat);
      break;
      case TRANS_COMPLETE:
      i++;
      double nf = run.next_finish_time();
      if (nf > current_time)
      {
        current_time = nf;
      }
      if (run.remove_transaction(&t))
      {
        wait.group_finished(t.group);
        total_transaction_execution_time += t.execution_time;
        total_transaction_time_in_system += current_time -
t.arrive_time;
      }
      break;
    }
    int number_running = run.n_running();
    int number_waiting = wait.n_wait();
    int number_in = number_running + number_waiting;
    if (number_in > 450)
    {
      done = 1;
      *overflow = 1;
      printf("\n %ld %d overflow\n", i, number_in);
      return;
    }
    while (run.processor_free())
    {
      if (wait.get_next(&t))
      {
      run.start_transaction(&t);
      }
      else
      {
      break;
      }
    }
  }

  long n_completed;
  long n_missed;
  run.get_results(&n_completed, &n_missed);
  double avg_processors = total_transaction_execution_time /
current_time;
```

```
  double avg_in = total_transaction_time_in_system / current_time;
  double x;

  switch (in->dependent_variable)
  {
    case P_INCOMPATIBLE:
      x = in->p_incompat;
      break;
    case ARRIVAL_RATE:
      x = in->arrival_rate;
      break;
    case EXECUTION_TIME:
      x = in->execution_time;
      break;
  }

  fprintf(fp, "%lf\t", x);
  fprintf(fp, "%lf\t", double(n_missed)/double(n_completed));
  fprintf(fp, "%lf\t%lf\t", avg_processors, avg_in);
  fprintf(fp, "%ld\t%ld\t", n_missed, n_completed);
  fprintf(fp, "\n");

  printf("%lf\t", x);
  printf("%lf\t", double(n_missed)/double(n_completed));
  printf("%lf\t%lf\t\n", avg_processors, avg_in);

}

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

void main(void)
{
  int i, j;
  compatible_t c;
  compat = &c;

  fp = fopen("output.txt", "w");
  if (fp == NULL)
  {
    perror("Can't open output.txt");
    exit(0);
  }

  input_data_t in;
  in.read();


  in.print(fp);
```

```
in.print(stdout);

fprintf(fp, "\n");
printf("\nHit any key to start ...\n\n");
getch();

switch (in.dependent_variable)
{
  case P_INCOMPATIBLE:
    printf("P. Incompatible\t");
    fprintf(fp, "P. Incompatible\t");
    break;
  case ARRIVAL_RATE:
    printf("Arrival Rate\t");
    fprintf(fp, "Arrival Rate\t");
    break;
  case EXECUTION_TIME:
    printf("Execution Time\t");
    fprintf(fp, "Execution Time\t");
    break;
}
printf("% Missed\tAvg. Proc.\tAvg. In\n");
fprintf(fp, "% Missed\tAvg. Proc.\tAvg. In\n");
double inc = (in.high - in.low) / in.n_inc;

for (double x=in.low; x < (in.high + (0.1 * inc)); x+= inc)
{
  if (kbhit())
  {
    /* Abort if they hit escape */
    if (getch() == 0x1B)
    {
    break;
    }
  }
  switch (in.dependent_variable)
  {
    case P_INCOMPATIBLE:
    in.p_incompat = x;
    break;
    case ARRIVAL_RATE:
    in.arrival_rate = x;
    break;
    case EXECUTION_TIME:
    in.execution_time = x;
    break;
  }
  int overflow;
```

```
    simulate(&in, &overflow);
    if (overflow)
    {
      break;
    }
  }
  fclose(fp);
}
```

```
/*****************************************************************
*                          TRANS.CPP                             *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*  This file contains the classes to manage the waiting          *
*  and running transactions                                      *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>

#include "compat.h"
#include "sim.h"
#include "trans.h"
#include "util.h"

compatible_t *compat;
double current_time;
extern unsigned _stklen = 62000;

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

transaction_t::transaction_t(void)
{
  priority = 0;
  group = 0;
  num_functions = 0;
  num_waiting = 0;
  deadline = 0;
  execution_time = 0;
  finish_time = 0;
}

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

waiting_t::waiting_t(void)
{
  n=0;
}

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

void waiting_t::reset(void)
```

```
{
  n=0;
}

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

int waiting_t::num_compare(int nf)
{
  int num = 0;
  for (int i=0; i<n; i++)
  {
    num += nf * w[i].num_functions;
  }
  return num;
}

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

void waiting_t::print_all(void)
{
  for (int i=0; i < n; i++)
  {
    printf("(%d,%d)", w[i].group, w[i].num_waiting);
  }
}

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

void waiting_t::group_finished(int g)
{
  for (int i=0; i < n; i++)
  {
    if (!compat->get(g, w[i].group))
    {
      w[i].num_waiting--;
    }
  }
}

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

void waiting_t::insert_transaction(transaction_t *t, double p)
{
  if (n > 499)
  {
    printf("Too many transactions\n");
    return;
```

```
  }
  for (int i=0; i < n; i++)
  {
    int is_compat = next_is_compatible(p);
    compat->set(w[i].group, t->group, is_compat);
    if (!is_compat)
    {
      t->num_waiting++;
    }
  }
  if (n >= MAX_WAIT)
  {
    printf("Error: %d\n", __LINE__);
    exit(0);
  }
  w[n] = *t;
  n++;
}

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

int waiting_t::get_next(transaction_t *t)
{
  int position = -1;
  for (int i=0; i < n; i++)
  {
    if (w[i].num_waiting == 0)
    {
      position = i;
      break;
    }
  }
  if (position == -1)
  {
    return 0;
  }
  *t = w[position];
  for (i=position; i<n-1; i++)
  {
    if (i >= MAX_WAIT)
    {
      printf("Error: %d\n", __LINE__);
      exit(0);
    }
    w[i] = w[i+1];
  }
  n--;
```

```
  return 1;
}

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

running_t::running_t(void)
{
  n = 0;
  n_completed = 0;
  n_missed = 0;
}

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

void running_t::reset(void)
{
  n = 0;
  n_completed = 0;
  n_missed = 0;
}

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

int running_t::num_compare(int nf)
{
  int num = 0;
  for (int i=0; i<n; i++)
  {
    num += nf * r[i].num_functions;
  }
  return num;
}

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

void running_t::print_all(void)
{
  for (int i=0; i < n; i++)
  {
    printf("(%d,%d)", r[i].group, r[i].num_functions);
  }
}

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

void running_t::start_transaction(transaction_t *t)
{
```

```
  if (n >= NCT)
  {
    return;
  }
  t->finish_time = current_time + t->execution_time;
  int position = 0;
  for (int i=n-1; i >= 0; i--)
  {
    if (t->finish_time < r[i].finish_time)
    {
      position = i+1;
      break;
    }
    if ((i+1) >= NCT)
    {
      printf("Error: %d\n", __LINE__);
      exit(0);
    }
    r[i+1] = r[i];
  }
  n++;
    if (position >= NCT)
    {
      printf("Error: %d\n", __LINE__);
      exit(0);
    }
  r[position] = *t;
}

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

int running_t::remove_transaction(transaction_t *t)
{
  if (n > 0)
  {
    n--;
    *t = r[n];
    n_completed++;
    if (t->finish_time > t->deadline)
    {
      n_missed++;
    }
    return 1;
  }
  return 0;
}

/*****************************************************************/
```

```
int running_t::incompatible_with(int group, double p)
{
  int cnt;
  cnt = 0;
  for (int i=0; i < n; i++)
  {
    int is_compat = next_is_compatible(p);
    compat->set(r[i].group, group, is_compat);
    if (!is_compat)
    {
      cnt++;
    }
  }
  return cnt;
}

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


double running_t::next_finish_time(void)
{
  if (n>0)
  {
    return r[n-1].finish_time;
  }
  else
  {
    return 0.0;
  }
}
```

```
/*****************************************************************
*                          UTIL.CPP                              *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*  This file contains utility programs to return random          *
* parameters for the simulation.                                 *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>

#include "util.h"
#include "random.h"

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

double get_next_arrival_time(double avg)
{
  return random_exponential(avg, 0);
}

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

double get_execution_time(int n, double avg)
{
  double extime = 0.0;
  for (int i=0; i < n; i++)
  {
    extime += random_exponential(avg, 1);
  }
  return extime;
}

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

int get_n_functions(double avg)
{
  int n = int(1.0 + random_exponential(avg - 0.5, 2));
  if (n < 1)
  {
    n = 1;
  }
  return n;
}

/*****************************************************************/
```

```
int current_group = 0;

int get_next_group()
{
  current_group = (current_group + 1) % 200;
  return current_group;
}

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

int next_is_compatible(double p)
{
  double x = Random(3);
  return (x > p);
}


/*****************************************************************/
```

```
/*****************************************************************
*                    RANDOM.CPP                                  *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
* This file containes procedures to generate 10 uniform random   *
* number streams. There are also procedures to change seeds.     *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int current_seed = 0;
long seed[10][10] =
{
  { 289,2063,2608,2539,2809,2645,9063,2875,1053,8201},
  {3447,4059,1289,5447,1609,3529,4243,4725,9025,7819},
  {2375,6167, 187,6019,1937,9465,9511,2957,9503, 683},
  {5913,8023,7669,4573,8285, 569,9581,5805,4025,7839},
  {7023,7567,7831,6337,1703, 963,2919,2619,4469,3649},
  {1893,4387,2107,1139,9409,7873,6439,9245,3865,2415},
  {5383,1039, 261,8197, 675,6825,1451,8047,1943, 529},
  {9957,9275,1811, 947,7547,6459,1725,1307,4729,1119},
  {3201,5561,7407,9245,4155,4845,2823,2561,5021,2573},
  {2027,4533,4355,1559,1027,6993,8587,1641,7807,8357},
};

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

void next_seeds(void)
{
  current_seed++;
  current_seed %= 10;
}

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

void first_seeds(void)
{
  current_seed = 0;
}

/*****************************************************************
This algorithm is from:
"The Art of Computer Systems Performance Analysis", page 443.
```

```
It can be tested using X(0) = 1, For a correct implementation,
X(10,000) should be 1,043,618,065.
*****************************************************************/
static const long a = 16807;
static const long m = 2147483647;
static const long q = 127773;
static const long r = 2836;

double Random(int i)
{
  long x_div_q, x_mod_q, x_new;
  long *x = &seed[current_seed][i];
  x_div_q = *x / q;
  x_mod_q = *x % q;
  x_new = (a * x_mod_q) - (r * x_div_q);
  if (x_new > 0)
  {
    *x = x_new;
  }
  else
  {
    *x = x_new + m;
  }
  return double(*x) / double(m);
}

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

double random_exponential(double average, int i)
{
  double u = Random(i);
  return (-1) * average * log(u);
}

/*****************************************************************/
```

```
/*****************************************************************
*            COMPAT.CPP                                          *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*  This file contains the procedures to store transaction        *
*  compatibility information. Once we determine if two           *
* transactions are compatible, this information must be stored.  *
*                                                                *
*****************************************************************/


#include <stdio.h>
#include <stdlib.h>

#include "compat.h"


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

void compatible_t::get_byte_bit(int n, int *byte, int *bit)
{
  *byte = n / 8;
  *bit = n % 8;
}

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

void compatible_t::set_bit(unsigned char *byte, int bit, int value)
{
  unsigned char mask = 0x01 << bit;
  if (value)
  {
    *byte |= mask;
  }
  else
  {
    mask = ~mask;
    *byte &= mask;
  }
}

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

int compatible_t::get_bit(unsigned char byte, int bit)
{
  unsigned char mask = 0x01 << bit;
  byte &= mask;
```

```
  return (byte != 0);
}

/*****************************************************************/
static const long a = 16807;
static const long m = 2147483647;
static const long q = 127773;
static const long r = 2836;


void compatible_t::set(int a, int b, int value)
{
  if ((a >= MAX_COMPAT) || (b >= MAX_COMPAT))
  {
    printf("Error %s %d\n", __FILE__, __LINE__);
    exit(0);
  }
  int byte, bit;

  get_byte_bit(a, &byte, &bit);
  set_bit(&c[b][byte], bit, value);
  get_byte_bit(b, &byte, &bit);
  set_bit(&c[a][byte], bit, value);
}

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

int compatible_t::get(int a, int b)
{
  if ((a >= MAX_COMPAT) || (b >= MAX_COMPAT))
  {
    printf("Error %s %d\n", __FILE__, __LINE__);
    exit(0);
  }
  int byte, bit;
  get_byte_bit(b, &byte, &bit);
  return get_bit(c[a][byte], bit);
}
/*****************************************************************/
```

```
/*****************************************************************
*                          INDATA.CPP                            *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
* This file contains routines to read and print input data.      *
*                                                                *
*****************************************************************/

#include <stdio.h>
#include <stdlib.h>

#include "sim.h"

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

void input_data_t::read(void)
{
  char line[200];
  FILE *fp = fopen("input.txt", "r");
  if (fp == NULL)
  {
    perror("Can't open input.txt");
    exit(0);
  }
  fgets(line, sizeof(line), fp);
  switch (line[0])
  {
    case 'P':
    case 'p':
      dependent_variable = P_INCOMPATIBLE;
      break;
    case 'A':
    case 'a':
      dependent_variable = ARRIVAL_RATE;
      break;
    case 'E':
    case 'e':
      dependent_variable = EXECUTION_TIME;
      break;
    default:
      printf("Invalid dependent variable: '%c'\n", line[0]);
      exit(0);
      break;
  }
  fgets(line, sizeof(line), fp);
  sscanf(line, "%lf", &low);
  fgets(line, sizeof(line), fp);
```

```
  sscanf(line, "%lf", &high);
  fgets(line, sizeof(line), fp);
  sscanf(line, "%ld", &n_inc);
  fgets(line, sizeof(line), fp);
  sscanf(line, "%lf", &arrival_rate);
  fgets(line, sizeof(line), fp);
  sscanf(line, "%lf", &execution_time);
  fgets(line, sizeof(line), fp);
  sscanf(line, "%lf", &n_functions);
  fgets(line, sizeof(line), fp);
  sscanf(line, "%lf", &deadline);
  fgets(line, sizeof(line), fp);
  sscanf(line, "%lf", &overhead);
  fgets(line, sizeof(line), fp);
  sscanf(line, "%lf", &p_incompat);
  fclose(fp);
}

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

void input_data_t::print(FILE *fp)
{
  fprintf(fp, "Independent Variable: ");
  switch (dependent_variable)
  {
    case P_INCOMPATIBLE:
      fprintf(fp, "Probability Incompatible\n");
      break;
    case ARRIVAL_RATE:
      fprintf(fp, "Arrival Rate\n");
      break;
    case EXECUTION_TIME:
      fprintf(fp, "Execution Time\n");
      break;
    default:
      fprintf(fp, "INVALID\n");
  }
  fprintf(fp, "Low\t\t%lf\n", low);
  fprintf(fp, "High\t\t%lf\n", high);
  fprintf(fp, "Number of Inc.\t\t%ld\n", n_inc);
  if (dependent_variable != ARRIVAL_RATE)
  {
    fprintf(fp, "Arrival Rate\t\t%lf\n", arrival_rate);
  }
  if (dependent_variable != EXECUTION_TIME)
  {
    fprintf(fp, "Execution Time (ms)\t\t%lf\n", 1000.0 *
execution_time);
```

```
  }
  fprintf(fp, "Number of Func.\t\t%lf\n", n_functions);
  fprintf(fp, "Deadline (ms)\t\t%lf\n", 1000 * deadline);
  fprintf(fp, "Overhead (ns)\t\t%lf\n", 1.0e9 * overhead);
  if (dependent_variable != P_INCOMPATIBLE)
  {
    fprintf(fp, "Prob. Incompatible\t\t%lf\n", p_incompat);
  }
}

/*****************************************************************/
```

```
/*********************************************************************
*                          SIM.H                                     *
*                                                                    *
* Author: Don W. Carr                                                *
*                                                                    *
*  Simulation header file                                            *
*                                                                    *
*********************************************************************/

#ifndef __SIM_INC__
#define __SIM_INC__

#define P_INCOMPATIBLE (0)
#define ARRIVAL_RATE (1)
#define EXECUTION_TIME (2)

class input_data_t
{
public:
  int dependent_variable;
  double low;
  double high;
  long n_inc;
  double arrival_rate;
  double execution_time;
  double n_functions;
  double deadline;
  double overhead;
  double p_incompat;
  void read(void);
  void print(FILE *fp);
};

#endif
```

```
/*****************************************************************
*                          TRANS.H                               *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*  Transaction header file                                       *
*                                                                *
*****************************************************************/

#ifndef __TRANS_INC__
#define __TRANS_INC__

#define NPROC (8)
#define NCT (NPROC-1)
#define MAX_WAIT (1000)
#define TRANS_COMPLETE (0)
#define TRANS_ARRIVE (1)


class transaction_t
{
public:
  int priority;
  int group;
  int num_functions;
  int num_waiting;
  double deadline;
  double execution_time;
  double arrive_time;
  double finish_time;
  transaction_t(void);
};

class running_t
{
private:
  int n;
  long n_completed;
  long n_missed;
  transaction_t r[NCT];

public:
  running_t(void);
  void reset(void);
  void print_all(void);
  int num_compare(int nf);
  void get_results(long *n_complete, long *n_miss) {
     *n_complete = n_completed; *n_miss = n_missed;};
```

```
  int n_running(void) {return n;};
  double next_finish_time(void);
  int incompatible_with(int group, double p);
  void start_transaction(transaction_t *t);
  int remove_transaction(transaction_t *t);
  int processor_free(void) {return (n < NCT);};
};

class waiting_t
{
private:
  int n;
  transaction_t w[MAX_WAIT];

public:
  waiting_t(void);
  void reset(void);
  void print_all(void);
  int num_compare(int nf);
  int n_wait(void) {return n;};
  int get_next(transaction_t *t);
  void group_finished(int g);
  void insert_transaction(transaction_t *t, double p);
};

extern double current_time;
extern compatible_t *compat;

#endif
```

```
/*********************************************************************
*                        UTIL.H                                      *
*                                                                    *
* Author: Don W. Carr                                                *
*                                                                    *
*   Utility header file                                              *
*                                                                    *
*********************************************************************/
#ifndef __UTIL__INC__
#define __UTIL__INC__

double get_next_arrival_time(double avg);
double get_execution_time(int n, double avg);
int get_n_functions(double avg);
int get_next_group();
int next_is_compatible(double p);

#endif
```

```
/*****************************************************************
*                          COMPAT.H                              *
*                                                                *
* Author: Don W. Carr                                            *
*                                                                *
*   Header file for transaction compatibility class.             *
*                                                                *
*****************************************************************/

#ifndef __COMPAT_INC__
#define __COMPAT_INC__

#define MAX_COMPAT (500)

class compatible_t
{
private:
  unsigned char c[MAX_COMPAT][(MAX_COMPAT/8) + 1];
  void get_byte_bit(int n, int *byte, int *bit);
  void set_bit(unsigned char *byte, int bit, int value);
  int get_bit(unsigned char byte, int bit);
public:
  void set(int a, int b, int value);
  int get(int a, int b);
};

#endif
```

```
/*********************************************************************
*                          RANDOM.H                                  *
*                                                                    *
* Author: Don W. Carr                                                *
*                                                                    *
*  Header file for random generator routines.                        *
*                                                                    *
*********************************************************************/

#ifndef __RANDOM_INC__
#define __RANDOM_INC__

double Random(int i);
double random_exponential(double average, int i);
void first_seeds(void);
void next_seeds(void);

#endif
```

APPENDIX E. SIMULATOR OUTPUT DATA

```
Independent Variable: Arrival Rate
Low          675.000000
High         13500.000000
Number of Inc.          19
Execution Time (ms)          0.125000
Number of Func.         4.000000
Deadline (ms)           2.000000
Overhead (ns)           10.000000
Prob. Incompatible           0.050000

Arrival Rate      % Missed    Avg. Proc.  Avg. In
  675.000000          0.020300          0.336395      0.343131
 1350.000000          0.021000          0.689025      0.715369
 2025.000000          0.023500          1.012028      1.064651
 2700.000000          0.026300          1.369325      1.481057
 3375.000000          0.028200          1.688750      1.859391
 4050.000000          0.029100          2.038381      2.309119
 4725.000000          0.030100          2.294034      2.639055
 5400.000000          0.039200          2.750829      3.296951
 6075.000000          0.039900          3.040715      3.702761
 6750.000000          0.041600          3.361339      4.190058
 7425.000000          0.042700          3.643580      4.655890
 8100.000000          0.057000          4.047233      5.586699
 8775.000000          0.056900          4.456347      6.386447
 9450.000000          0.084400          4.698608      7.856905
10125.000000          0.103700          5.100124      9.053482
10800.000000          0.116200          5.454531      10.306129
11475.000000          0.210100          5.883021      14.579418
12150.000000          0.306000          6.212882      19.116604
12825.000000          0.337300          6.384402      21.606518
overflow
```

Independent Variable: Arrival Rate
Low 675.000000
High 13500.000000
Number of Inc. 19
Execution Time (ms) 0.125000
Number of Func. 4.000000
Deadline (ms) 2.000000
Overhead (ns) 10.000000
Prob. Incompatible 0.080000

Arrival Rate	% Missed	Avg. Proc.	Avg. In
675.000000	0.020900	0.336400	0.346351
1350.000000	0.023900	0.666922	0.708612
2025.000000	0.026300	1.006009	1.107741
2700.000000	0.030400	1.349439	1.544241
3375.000000	0.035600	1.717471	2.046212
4050.000000	0.039900	2.013077	2.461070
4725.000000	0.052900	2.375244	3.094196
5400.000000	0.051200	2.669186	3.666908
6075.000000	0.068900	3.054990	4.458032
6750.000000	0.104600	3.403029	5.803189
7425.000000	0.095400	3.648335	6.060844
8100.000000	0.126900	3.997581	7.583214
8775.000000	0.174800	4.386011	9.904985
9450.000000	0.264700	4.812044	13.546173
10125.000000	0.509000	5.147225	32.025761
overflow			

```
Independent Variable: Arrival Rate
Low          675.000000
High         13500.000000
Number of Inc.           19
Execution Time (ms)            0.125000
Number of Func.          4.000000
Deadline (ms)            2.000000
Overhead (ns)            10.000000
Prob. Incompatible             0.120000

Arrival Rate       % Missed    Avg. Proc.  Avg. In
  675.000000           0.022100          0.336350      0.351735
 1350.000000           0.027200          0.689010      0.754463
 2025.000000           0.032300          1.006778      1.153014
 2700.000000           0.043300          1.367144      1.703601
 3375.000000           0.050600          1.684512      2.230088
 4050.000000           0.070500          2.032669      3.022832
 4725.000000           0.096600          2.370245      4.155313
 5400.000000           0.119000          2.693365      5.045265
 6075.000000           0.159600          2.995662      6.385547
 6750.000000           0.249300          3.313738      9.162597
 7425.000000           0.424600          3.713309      17.425937
overflow


Independent Variable: Arrival Rate
Low          675.000000
High         13500.000000
Number of Inc.           19
Execution Time (ms)            0.125000
Number of Func.          4.000000
Deadline (ms)            2.000000
Overhead (ns)            10.000000
Prob. Incompatible             0.150000

Arrival Rate       % Missed    Avg. Proc.  Avg. In
  675.000000           0.024100          0.336348      0.357796
 1350.000000           0.030200          0.685126      0.766261
 2025.000000           0.036900          0.999547      1.213522
 2700.000000           0.058000          1.385369      1.873104
 3375.000000           0.062400          1.667026      2.385700
 4050.000000           0.091100          2.008440      3.329147
 4725.000000           0.169000          2.403254      5.211511
 5400.000000           0.200700          2.675452      6.445388
 6075.000000           0.381200          2.949837      12.150761
overflow
```

```
Independent Variable: Arrival Rate
Low          675.000000
High         13500.000000
Number of Inc.           19
Execution Time (ms)            0.125000
Number of Func.          4.000000
Deadline (ms)            2.000000
Overhead (ns)            10.000000
Prob. Incompatible             0.200000

Arrival Rate       % Missed    Avg. Proc.  Avg. In
  675.000000           0.025300          0.336400      0.362004
 1350.000000           0.033500          0.674807      0.779841
 2025.000000           0.048000          1.013318      1.344562
 2700.000000           0.078500          1.363333      2.108701
 3375.000000           0.105400          1.674068      2.950071
 4050.000000           0.246000          2.081686      6.074090
 4725.000000           0.357200          2.328505      9.068558
overflow

Independent Variable: Arrival Rate
Low          100.000000
High         13520.130000
Number of Inc.           10
Execution Time (ms)            0.125000
Number of Func.          4.000000
Deadline (ms)            1.000000
Overhead (ns)            10.000000
Prob. Incompatible             0.050000

Arrival Rate       % Missed    Avg. Proc.  Avg. In
  100.000000           0.134000          0.049825      0.049944
 1442.013000           0.148500          0.719427      0.751611
 2784.026000           0.158000          1.401156      1.507508
 4126.039000           0.175700          2.073036      2.371422
 5468.052000           0.184700          2.732407      3.222235
 6810.065000           0.205700          3.381370      4.236405
 8152.078000           0.245900          4.137878      5.865823
 9494.091000           0.300700          4.902002      7.986511
10836.104000           0.348600          5.276331      10.253996
12178.117000           0.535600          6.118069      17.891748
13520.130000           0.841900          6.728642      40.564886
```

Independent Variable: Arrival Rate
Low 100.000000
High 13520.130000
Number of Inc. 10
Execution Time (ms) 0.125000
Number of Func. 4.000000
Deadline (ms) 2.000000
Overhead (ns) 10.000000
Prob. Incompatible 0.050000

Arrival Rate	% Missed	Avg. Proc.	Avg. In
100.000000	0.018900	0.049825	0.049944
1442.013000	0.023100	0.719427	0.751611
2784.026000	0.022900	1.401156	1.507508
4126.039000	0.032300	2.073036	2.371422
5468.052000	0.033600	2.732407	3.222235
6810.065000	0.038800	3.381370	4.236405
8152.078000	0.065800	4.137878	5.865823
9494.091000	0.089200	4.902002	7.986511
10836.104000	0.127500	5.276331	10.253996
12178.117000	0.242700	6.118069	17.891748
13520.130000	0.656900	6.728642	40.564886

Independent Variable: Arrival Rate
Low 100.000000
High 13520.130000
Number of Inc. 10
Execution Time (ms) 0.125000
Number of Func. 4.000000
Deadline (ms) 4.000000
Overhead (ns) 10.000000
Prob. Incompatible 0.050000

Arrival Rate	% Missed	Avg. Proc.	Avg. In
100.000000	0.000200	0.049825	0.049944
1442.013000	0.000100	0.719427	0.751611
2784.026000	0.001000	1.401156	1.507508
4126.039000	0.001700	2.073036	2.371422
5468.052000	0.000700	2.732407	3.222235
6810.065000	0.001100	3.381370	4.236405
8152.078000	0.003800	4.137878	5.865823
9494.091000	0.007200	4.902002	7.986511
10836.104000	0.016500	5.276331	10.253996
12178.117000	0.053100	6.118069	17.891748
13520.130000	0.301800	6.728642	40.564886

Independent Variable: Execution Time
Low 0.000010
High 0.000140
Number of Inc. 10
Arrival Rate 12000.000000
Number of Func. 4.000000
Deadline (ms) 2.000000
Overhead (ns) 10.000000
Prob. Incompatible 0.050000

Execution Time	% Missed	Avg. Proc.	Avg. In
0.000010	0.000000	0.470770	0.484972
0.000023	0.000000	1.132610	1.216033
0.000036	0.000000	1.724615	1.896875
0.000049	0.000200	2.410441	2.795856
0.000062	0.001200	3.006940	3.633537
0.000075	0.006300	3.499448	4.484866
0.000088	0.016900	4.232046	5.846315
0.000101	0.048700	4.910954	8.081933
0.000114	0.102900	5.415279	10.946041
0.000127	0.298000	6.182564	19.416937
0.000140	0.827400	6.698317	64.195190

APPENDIX F. UPDATE METHODS

In this appendix we discuss methods for updating dependent objects. The problem arises when a rule object is dependent on values from one or more other objects. With REACT, rule objects are notified of changes by other objects through the encapsulated event system. For the prototype system, all rule objects re-calculate (or take actions on other objects) immediately on receipt of event notifications. For many applications this is acceptable, but for completeness we will discuss other options, and discuss some of the problems with this method. The four basic update methods we will discuss are: 1) event-driven, 2) demand-driven, 3) periodic, and 4) scan sequence. To illustrate these update methods the example in Figure X will be used. The objects A, B, C, D, E, and F are input interface objects, object G is a rule object that derives its value from A, B, and C, object H is a rule object that derives its value from D, E, and F, and object I is a rule object that derives its value from objects G and H.

Event-Driven Updates

With event-driven updates, rule objects immediately update internal values upon receiving event notifications. This is currently the update method supported for the REACT database kernel prototype. As an example, see Figure F.1. If object A is

updated, this will cause rule object G to update, and this in turn will cause rule object I to update. If B is updated, this will cause rule object G to update, and this in turn will cause rule object I to update. Similar actions would occur when C, D, E, and F are updated.

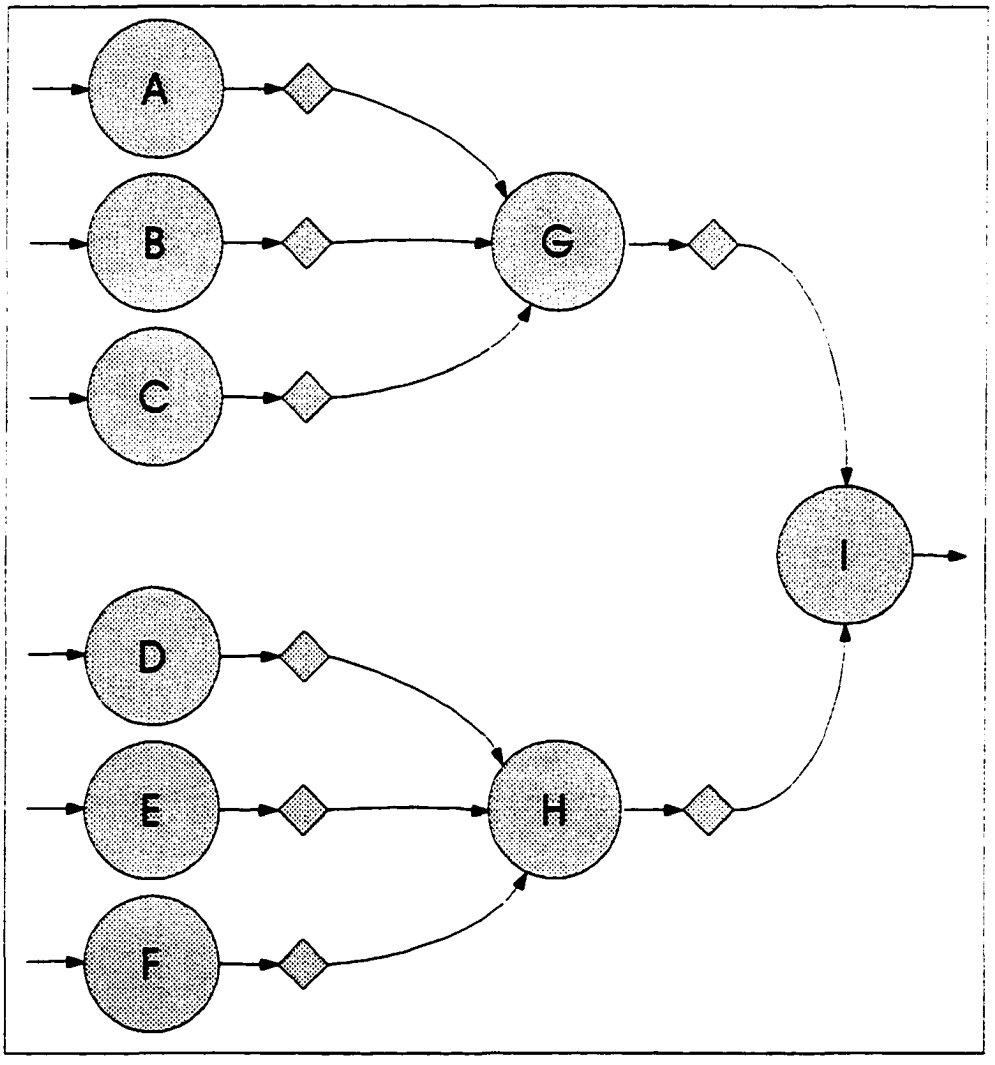

Figure F.1 Update Example

Demand-Driven Updates

With demand-driven updates, objects are updated only when there is a demand for the dependent values, possibly as the result of a query. As an example assume that we query object I in Figure X. This would require us to first get the values from G and H, and in turn the values from A, B, C, D, E, and F. We would then recalculate the values for G and H, and then finally recalculate the value for I, and then return the results of the query. As you can see this could be quite inefficient if there were not some way of determining if values have changed since the object was last updated. There would need to be some way to determine quickly if dependent objects had changed since the last recalculation. Also, for most real-time systems, most rules would always need to be evaluated immediately in order to trigger alarms or send outputs in response to events.

Periodic Updates

With periodic updates, objects are evaluated periodically without regard to the time that other objects have been updated. With this method of update, objects are evaluated at regular intervals regardless of whether dependent values have changed. In this scenario, if we query object I in Figure X, the value from the last update is

used. Objects G, H, and I would be periodically updated at some interval. It would, of course, be desirable to update G and H before updating I. We mention this method for completeness only, since the temporal consistency of objects is compromised. This method has been widely used in the past because of it's simplicity and ease of implementation.

Scan Sequence Updates

Scan sequence is a modified event driven update method that substantially improves performance and eliminates the possibility of mismatched data used in calculations. With scan sequence updates we take a closer look at the way objects are updated from the real-world. Typically, a field device will scan input values at regular intervals in order that database objects can be updated to reflect values in the real-world. If one rule object receives event notifications from, and is therefore dependent on, multiple objects updated from the same real-world device, then it is desirable that the dependent rule object is only evaluated once each time the real-world device is scanned. In Figure A.1, we assume that all values are scanned from the same real-world device, if we use a strict event-driven policy, then each time the real-world device is scanned, one value changed event will be generated by the interface objects A, B, C, D, E, and F, three value changed events will be generated by G and H, and six value changed events will be generated by I. There are two basic problems with

this approach: 1) Extra CPU time is used to perform twelve rule evaluations and actions (G three times, H three times, and I six times) when only three are required, 2) Imprecise calculations can result when values that are not from the same device scan are used to perform a calculation. With scan sequence we assign a sequence number to each device scan and only update dependent rule objects when value changed events have been received with the same sequence number for all dependent objects. In other words, rule objects would delay executing update member functions, triggering higher level events, or updating internal values until all events from a particular device scan have been received. The one disadvantage is that this will probably require some extra work on the part of the object programmer. The system can generate an extra event such as "scan complete" when all events have been delivered from a particular device, but the object programmer would have to write an event handler for this event and delay calculations until this event arrives. We could also deliver all events at the same time, but the object programmer would still have to modify event handlers to account for this. It is desirable to minimize the impact on the object programmer as much as possible.

Summary

The strict event-driven approach is the most intuitive update model to use. With this model, the rule object programmer is only responsible for specifying actions

when events arrive and need not worry about how data arrives from real-world devices. With the scan sequence model, we avoid redundant and mis-matched calculations, but we place the burden of understanding how data arrives from the real-world on the rule object programmer.

BIBLIOGRAPHY

[1] R. Abbott and H. Molina: "Scheduling Real-Time Transactions: A Performance Evaluation". *ACM Transactions on Database Systems*, Volume 17, Number 3, pp. 46-52, Sept. 1992.

[2] R. Abbott and H. Molina: "Scheduling Real-Time Transactions". *SIGMOD Record*, Volume 17, Number 1, pp. 71-81, March 1988.

[3] B. Adelberg, B. Kao, and H. Garcia-Molina: "Overview of the Stanford Real-time Information Processor (STRIP)". *SIGMOD Record*, Volume 25, Number 1, pp. 34-37, March 1996.

[4] R. Agrawal, S. Dar, and N. Gehani: "The O++ Database Programming Language". *IEEE Computer Ninth International Conference on Data Engineering*, pp. 55-61, Vienna, Austria, Apr. 1993.

[5] R. Agrawal and N. Gehani: "ODE (Object Database and Environment): The Language and the Data Model". *SIGMOD Record* Volume 18, Number 1, pp. 152-161, March 1989.

[6] H. Agusleo and S. Soparkar: "Logic-Enhanced Memory Database: A Simulation Study". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 6 pages, March 7-8, 1996.

[7] A. Aho, J. Hopcroft, and J. Ullman: *Data Structures and Algorithms*, The Addison-Wesley Publishing Company, Reading, Massachusetts, 1992.

[8] A. Aiken, J. Hellerstein, and J. Widom: "Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism". *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, pp. 59-68, June 1992.

[9] A. Aiken, J. Hellerstein, and J. Widom: "Static Analysis Techniques for Predicting the Behavior of Active Database Rules". *ACM Transactions on Database Systems*, Volume 20, Number 1, pp. 3-41, March 1995.

[10] A. Albano, R. Bergammini, G. Ghelli, and R. Orsini: "An Object Data Model with Roles". *19th International Conference on Very Large Databases*, Dublin, Ireland, pp. 39-51, Aug. 1993.

[11] J. Anderson, S. Ramamurthy, M. Moir, and K. Jeffay: "Lock-Free Transactions for Real-Time Systems". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[12] S. Andler, J. Hasson, J. Eriksson, J. Mellin, M. Berndtsson, and B. Eftring: "DeeDS Towards a Distributed and Active Real-Time Database System". *SIGMOD Record*, Volume 25, Number 1, pp. 38-40, March 1996.

[13] E. Anwar, L. Maugis, and S. Chakravarthy: "A New Perspective on Rule Support for Object-Oriented Databases". *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, Washington, DC, pp. 99-108, May 1993.

[14] A. Asthana, H. Jagadish, and P. Krzyzanowski: "Architectural Support For Real-Time Database Systems". *Technical Report*, Department of Computer Sciences, The University of Texas at Austin, TR-92-16, April 1992.

[15] M. Berndtsson and J. Hansson: "Workshop Report: The First International Workshop on Active and Real-Time Database Systems (ARTDB-95)". *SIGMOD Record*, Volume 25, Number 1, pp. 64-66, March 1996.

[16] E. Bertino and L. Martino: *Object-Oriented Database Systems*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1993.

[17] A. Bestavros and S. Nagy: "Value-cognizant Admission Control Strategies for Real-Time DBMS". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[18] A. Bestavros, K. Lin, and S. Son: "Report on First International Workshop on Real-Time Database Systems". *SIGMOD Record*, Volume 25, Number 3, pp. 50-52, September 1996.

[19] A. Bestavros: "Advances in Real-Time Database Systems Research". *SIGMOD Record*, Volume 25, Number 1, pp. 3-7, March 1996.

[20] G. Booch: *Object-oriented Analysis and Design with Applications (Second Edition)*, The Benjamin/Cummings Publishing Company, Menlo Park, California, 1994.

[21] H. Branding and A. Buchmann: "Unbundling RTDBMS functionality to support WWW-Applications". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 3 pages, March 7-8, 1996.

[22] K. Brathwaite: *Object-Oriented Database Design, Concepts and Application*. Academic Press, New York, New York, 1993.

[23] G. Bulzingsloewen, K. Dittrich, C. Iochpe, R. Liedke, P. Lockemann, and M. Schryro: "KARDAMOM - A Dataflow Database Machine for Real-Time Applications". *SIGMOD Record*, Volume 17, Number 1, pp. 44-50, March 1988.

[24] J. Byun, A. Burns, and A. Wellings: "A Worst-Case Behaviour Analysis for Hard Real-Time Transactions". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 6 pages, March 7-8, 1996.

[25] H. Callison: "A Periodic Object Model for Real-Time Systems". *Technical Report*, Department of Computer Science and Engineering, University of Washington, TR-93-05-02, 1993.

[26] D. Carr, L. Miller, and T. Smay: "A Real-time System for High Transaction Loads and Moderate Deadlines". *12th International Conference on Computers and Their Applications*, Tempe, Arizona, pp. 134-139, March 1997.

[27] R. Cattell: *The Object Database Standard: ODMG - 93 (Release 1.1)*, Morgan Kaufmann Publishers, San Francisco, California, 1994.

[28] S. Ceri: "A Declarative Approach to Active Databases". *IEEE Computer Eighth International Conference on Data Engineering*, Tempe, Arizona, pp. 452-456, Feb. 1992.

[29] S. Chakravarthy, E. Anwar, and L. Maugis: "Design and Implementation of Active Capability for an Object-Oriented Database" *Technical Report*, Department of Computer and Information Sciences, University of Florida, UF-CIS-TR-93-001.

[30] S. Chakravarthy and D. Mishra: "Snoop: An Expressive Event Specification Language For Active Databases" *Technical Report*, Department of Computer and Information Sciences, University of Florida, UF-CIS-TR-93-007.

[31] H. Cheng and G. Ozsoyoglu: "A Framework for Cooperative Real-Time Transactions". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 4 pages, March 7-8, 1996.

[32] S. Dar, N. Gehani, H. Jagadish: "CQL++, A SQL for the Ode Object-Oriented Database". *Technical Report*, AT&T Bell Laboratories, Murray Hill, New Jersey, 1994.

[33] C. Date: *An Introduction to Database Systems (Fifth Edition)*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[34] U. Dayal, A. Buchmann, U. Chakravarthy, M. Hsu, R. Ledin, D. McCarthy, A. Rosenthal, and S. Sarin: "The HiPAC Project: Combining Active Databases and Timing Constraints". *SIGMOD Record*, Volume 17, Number 1, pp. 51-70, March 1988.

[35] U. Dayal and G. Wuu: "A Uniform Approach to Processing Temporal Queries". *18th International Conference on Very Large Databases*, Vancouver, Canada, pp. 407-418, Aug. 1992.

[36] M. Eich: "A Classification and Comparison of Main Memory Database Recovery Techniques". *Parallel Architectures for Database Systems*, pp. 417-424, IEEE Computer Society Press, 1989.

[37] E. Falkenroth, Peter Loborg, and A. Torne: "Databases in Control and Simulation". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 5 pages, March 7-8, 1996.

[38] N. Gal-Oz, E. Gudes, and E. Fernandez: "A Model of Methods Access Authorization in Object-oriented Databases". *19th International Conference on Very Large Databases*, Dublin, Ireland, pp. 52-61, Aug. 1993.

[39] E. Gamma, R. Helm, R. Johnson, and J. Vlissides: Design Patterns, Elements of Reusable Object-oriented Software, Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[40] H. Garcia-Molina, R. Lipton, and J. Valdes: "A Massive Memory Machine". *Parallel Architectures for Database Systems,* pp. 408-416, IEEE Computer Society Press, 1989.

[41] N. Gehani, H. Jagadish, and O. Shmueli: "Event Specification in an Active Object-Oriented Database". *ACM SIGMOD*, pp. 81-90, 1992.

[42] N. Gehani, H. Jagadish, and O. Shmueli: "Composite Event Specification in Active Databases: Model & Implementation". *18th International Conference on Very Large Databases,* Vancouver, Canada, pp. 327-338, Aug. 1992.

[43] P. Gopinath, R. Ramnath, and K. Schwan: "Data Base Design for Real-Time Adaptations". *Journal of Systems Software,* pp. 155-166, 1992.

[44] V. Gottemukkala and T. Lehman: "Locking and Latching in a Memory-Resident Database System". *18th International Conference on Very Large Databases,* Vancouver, Canada, pp. 533-544, Aug. 1992.

[45] M. Graham: "Issues in Real-Time Data Management". *Technical Report,* Software Engineering Institute, Carnegie Mellon University, CMU/SEI-91-TR-17, July 1991.

[46] W. Haque: "Transaction processing in real-time database systems". Ph.D. Dissertation, Iowa State University, Ames, Iowa, 1993.

[47] J. Haritsa and S. Seshadri: "Real-Time Index Concurrency Control". *SIGMOD Record,* Volume 25, Number 1, pp. 13-17, March 1996.

[48] M. He, L. Miller, A. Hurson, D. Sheth: *An Efficient Storage Protocol for Distributed Object Oriented Databases,* Fifth IEEE Symposium on Parallel and Distributed Processing. pp 606-610.

[49] D. Hong, S. Chakraverthy, and T. Johnson: "Locking Based Concurrency Control for Integrated Real-Time Database Systems". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 6 pages, March 7-8, 1996.

[50] J. Huang and L. Gruenwald: "An Update-Frequency-Valid-Interval Partition Checkpoint Technique for Real-Time Main Memory Databases". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 9 pages, March 7-8, 1996.

[51] J. Hughes: *Object Oriented Databases*. Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[52] S. Hung and K. Lam: "Locking Protocols for Concurrency in Real-Time Database Systems". *SIGMOD Record*, Volume 21, Number 4, pp. 22-27, Dec. 1992.

[53] H. Ishikawa: *Object-Oriented Database System, Design and Implementation for Advanced Applications*. Springer-Verlag, Tokyo, Japan, 1993.

[54] H. Ishikawa and K. Kubota: "An Active Object-Oriented Database: A Multi-Paradigm Approach to Constraint Management". *19th International Conference on Very Large Databases*, Dublin, Ireland, pp 467-478, Aug. 1993.

[55] R. Jain: *The Art of Computer Systems Performance Analysis, Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, New York, New York, 1991.

[56] H. Jagadish, A. Silberschatz, S. Sudarshan: "Recovering from Main-Memory Lapses". *19th International Conference on Very Large Databases*, pp. 391-404, Dublin, Ireland, Aug. 1993.

[57] The Java Beans 1.0 API Specification. Sun Microsystems Inc., Palo Alto, California, 1996.

[58] A Joint Report by the ACT-NET Consortium: "The Active Database Management System Manifesto: A Rulebase of ADBMS Features". *SIGMOD Record*, Volume 25, Number 3, pp. 40-49, September 1996.

[59] A. Kemper and D. Kossmann: "Adaptable Pointer Swizzling Strategies in Object Bases". *IEEE Computer Ninth International Conference on Data Engineering*, Vienna, Austria, pp. 155-162, Apr. 1993.

[60] W. Kim: "Observations on the ODMG-93 Proposal for an Object-Oriented Database Language". Submitted to News Group, 1994.

[61] W. Kim: Object-Oriented Database Systems: "Promises, Reality, and Future". *19th International Conference on Very Large Databases*, Dublin, Ireland, pp. 676-687, Aug. 1993.

[62] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft and R. Zainlinger: "Distributed Fault-Tolerant Real-Time Systems: The Mars Approach". *IEEE MICRO*, pp. 25-40, February 1989.

[63] V. Lee, K. Lam, B. Kao, K. Lam, and S. Hung: "Priority Assignment for Sub-transaction in Distributed Real-time Databases". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[64] K. Lin and C. Peng: "Enhancing External Consistency in Real-Time Transactions". *SIGMOD Record*, Volume 25, Number 1, pp. 26-28, March 1996.

[65] K. Lin and M. Lin: "Enhancing Availability in Distributed Real-Time Databases". *SIGMOD Record*, Volume 17, Number 1, pp. 34-43, March 1988.

[66] S. Listgarten and M. Neimat: "Modeling Costs for a MM-DBMS". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 7 pages, March 7-8, 1996.

[67] L. Liu and R. Meersman: Activity Model: "A Declarative Approach for Capturing Communication Behavior in Object-Oriented Databases". *18th International Conference on Very Large Databases*, pp. 481-493, Vancouver, Canada, Aug. 1992.

[68] V. Lortz: "An Object-Oriented Real-Time Database System for Multiprocessors". Ph.D. dissertation, The University of Michigan, Ann Arbor, Michigan, 1994.

[69] M. Maekawa, A. Oldehoeft, and R. Oldehoeft: *Operating Systems Advanced Concepts*, The Benjamin/Cummings Publishing Company, Menlo Park, California, 1987.

[70] D. McCarthy and U. Dayal: "The Architecture of an Active Database Management System". *SIGMOD Record* Volume 18, Number 1, pp. 215-224, March 1989.

[71] J. Mellin, J. Gansson, and S. Andler: "Refining Design Constraints using a System Services Model of a Real-Time DBMS". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[72] L. Miller, a. Hurson, S. Pakzad, and H. Tsai: "Incorporating Time into the Associative Search Language Machine (ASLM)". *Parallel Architectures for Database Systems*, pp. 464-473, IEEE Computer Society Press, 1989.

[73] T. Padron-McCarthy and T. Tisch: "Performance-Polymorphic Execution of Real-Time Queries". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 4 pages, March 7-8, 1996.

[74] C. Peng, K. Lin, and C. Boettcher: "Real-Time Database Benchmark for Avionics Systems". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[75] J. Peterson and A. Sibershatz: *Operating System Concepts (Second Edition)*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1985.

[76] POET Release 2.0 Object-Oriented Database, *Users Manual*, Poet Software Corporation, San Mateo, California, 1993.

[77] P. Poncelet, M. Teisseire, R. Cicchetti, and L. Lakhal: "Towards a Formal Approach for Object Database Design". *19th International Conference on Very Large Databases*, Dublin, Ireland, pp. 278-289, Aug. 1993.

[78] K. Ramamritham, R. Sivasankaran, J. Stankovic, D. Twosley, M. Xiong: "Integrating Temporal, Real-Time, and Active Databases". *SIGMOD Record*, Volume 25, Number 1, pp. 8-12, March 1996.

[79] K. Ramamritham: "Real-Time Databases". *Distributed and Parallel Databases*, Volume 1, Number 2, pp. 198-226, April 1993.

[80] B. Rao: *Object-Oriented Databases, Technology, Applications, and Products*. McGraw-Hill, New York, New York, 1994.

[81] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen: *Object-Oriented Modeling and Design.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[82] D. Rumpel, S. Venkata, K. Pandji, C. Liu, and N. Nagdy: "Real-Time Database for Power Systems Using Language Oriented Data Structure". *IEEE Transactions on Power Systems*, pp. 993-1000, Volume 5, Number 3, August 1990.

[83] M. Schiebe and S. Pferrer: *Real-Time Systems Engineering and Applications.* Kluwer Academic Publishers, Norwell, Massachusetts, 1992.

[84] L. Sha, R. Rajkumar, and J. Lehoczky: "Concurrency Control for Distributed Real-Time Databases". *SIGMOD Record*, Volume 17, Number 1, pp. 82-98, March 1988.

[85] M. Singhal: "Issues and Approaches to Design of Real-Time Database Systems". *SIGMOD Record*, Volume 17, Number 1, pp. 19-33, March 1988.

[86] R. Sivasankaran, J. Stankovic, D. Towsley, B. Purimetla, and K. Ramamritham: "Priority assignment in real-time active databases". *The VLDB Journal*, Volume 5, Number 1, pp. 19-34, January 1996.

[87] R. Snodgrass and I. Ahn: "A Taxonomy of Time in Databases". *Parallel Architectures for Database Systems*, pp. 443-453, IEEE Computer Society Press, 1989.

[88] R. Snodgrass: "The Temporal Query Language TQuel". *Parallel Architectures for Database Systems*, pp. 454-463, IEEE Computer Society Press, 1989.

[89] Sang H. Son: *Advances in Real-Time Systems.* Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[90] S. Son, R. David, and B. Thuraisingham: "Improving Timeliness in Real-Time Database Systems". *SIGMOD Record*, Volume 25, Number 1, pp. 29-33, March 1996.

[91] S. Son, Y. Kim, and R. Beckinger: MRDB: "A Multi-User Real-Time Database Testbed". *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, pp. 543-552, 1994.

[92] M. Squadrito, L. DiPippo, and V. Wolfe: "Towards Priority Ceilings in Object Based Semantic Real-Time Concurrency Control". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[93] J. Stankovic and W. Zhao: "On Real-Time Transactions". *SIGMOD Record*, Volume 17, Number 1, pp. 4-18, March 1988.

[94] B. Stroustrup: *The C++ Programming Language (2nd Ed.)*, Addison-Wesley Publishing Company, Reading, Massachusetts, 1991.

[95] O. Torbjornsen, S. Hvasshovd, and Y. Kim: "Towards Real-Time Performance in a Scalable, Continuously Available Telecom DBMS". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[96] S. Tseng, Y. Chin, and W. Yang: "Scheduling Value-Based Transactions in Real-Time Main-Memory Databases". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 7 pages, March 7-8, 1996.

[97] J. Ullman: *Principles of Database and Knowledge-Base Systems (Volume I)*, Computer Science Press, Potomac, Maryland, 1988.

[98] O. Ulusoy and A. Buchmann: "Exploiting Main Memory DBMS Features to Improve Real-Time Concurrency Control Protocols". *SIGMOD Record*, Volume 25, Number 1, pp. 23-25, March 1996.

[99] O. Ulusoy: "Current Research on Real-Time Databases". *SIGMOD Record*, Volume 21, Number 4, pp. 16-21, December 1992.

[100] P. Van Der Stok, J. Van Der Wal, and A. Aerts: "Modeling and Construction of Real-Time Database Schedulers". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[101] S. Vrbsky: "Approximate Query Processing of Temporal Data in Real-Time Databases". *Proceedings of the ISMM International Conference Intelligent Information Management Systems*, Washington, DC, pp. 38-38, June 1-3, 1994.

[102] K. Watkins: *Discrete Event Simulation in C*, McGraw-Hill International, London, England, 1993.

[103] S. White: "Pointer Swizzling Techniques for Object-Oriented Database Systems". Ph.D. dissertation, University of Wisconsin, Madison, Wisconsin, 1994.

[104] A. Wolski, J. Karvonen, and A. Puolakka: "The RAPID Case Study: "Requirements for and the design of a Fast-Response Database System". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 7 pages, March 7-8, 1996.

[105] M. Xiong, J. Stankovic, K. Ramamritham, D. Towsley, and R. Sivasankaran: "Maintaining Temporal Consistency: Issues and Algorithms". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, March 7-8, 8 pages, 1996.

[106] C. Yen and L. Miller: "Extending the Object-Relational Interface to Support an Extensible View System for Multidatabase Integration and Interoperation". *ISMM Conference on Intelligent Information Management Systems*, pp. 56-61, 1994.

[107] L. Zhou, K. Shin, E. Rundersteiner, and N. Soparkar: "Probabilistic Real-Time Data Access with Interval Constraints". *Conference Proceedings, First International Workshop on Real-Time Databases: Issues and Applications*. Newport Beach, California, 8 pages, March 7-8, 1996.

[108] L. Zhou, E. Rundensteiner, and K. Shin: Schema Evolution for Real-Time Object-Oriented Databases. *Technical Report*, Department of Electrical Engineering and Computer Science, The University of Michigan.

[109] C. Zimmermann and V. Cahill: "Raising the Cub, Distributed Real-Time Support in Tigger". *Technical Report*, Department of Computer Science, University of Dublin, TCD-CS-94-44, 1994.

[110] S. White and D. Dewitt: "A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies". *18th International Conference on Very Large Databases*, Vancouver, Canada, pp. 419-430, Aug. 1992.

[111] G. Wuu and U. Dayal: "A Uniform Model for Temporal Object-Oriented Databases" *IEEE Computer Eighth International Conference on Data Engineering*, Tempe, Arizona, pp. 584-593, Feb. 1992.