CMU-CS-82-154

Monitoring Distributed Systems: A Relational Approach

Richard Snodgrass

December, 1982

DEPARTMENT
of
COMPUTER SCIENCE

Carnegie-Mellon University

CMU-CS-82-154

Monitoring Distributed Systems: A Relational Approach

Richard Snodgrass

December, 1982

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, PA 15213

Submitted to Carnegie-Mellon University in partial fulfillment of the requirements for the degree of Doctor Of Philosophy.

Copyright © 1982 Richard T. Snodgrass

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory Under Contract F33615-78-C-1551, in part by the Ballistic Missile Defense Advanced Technological Center Under Contract DASG60-81-0077, and in part by an NSF fellowship.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the BMDATC, the US Government, or Carnegie-Mellon University.

Ada is a registered trademark of the Department of Defense (Ada Joint Program Office). Unix is a registered trademark of Bell Laboratories.

Abstract

Monitoring is the extraction of dynamic information concerning a computational process, as that process executes. Distributed systems, with their qualitative and quantitative increases in complexity, demand an intelligent monitor. The thesis of this dissertation is that monitoring is fundamentally an information processing activity, and that the relational model, as applied in relational databases, is an appropriate formalization of this information. In this approach, the notions of entity (data structures, processes, hardware components, etc.) and relationship (processes running on processors, messages in queues, etc.) are structured as a set of time-varying relations. Queries on this collection of relations are translated into retrieval and computational activities to be performed by the monitor.

Data collection is an important aspect of monitoring. After discussing a model of the environment where data collection takes place, a flexible, strongly typed, efficient data collection mechanism is presented. The impact of various features of the environment on this mechanism is examined.

The user specifies the desired actions of the monitor with a high-level, non-procedural query language called TQuel. This language is a superset of Quel, a relational database query language, with additional syntax and semantics to incorporate time as an integral part of the language. A formal semantics with several useful properties relating to monitoring is presented.

Queries in TQuel must be converted into a procedural form in order to execute efficiently. Update networks, designed for dynamic, incremental updating of derived relations, are introduced as the target language for the TQuel translator. These structures are composed of access nodes, which interface effectively with the system being monitored, and operator nodes, which carry out the desired computations. The generation of update networks involves several sophisticated techniques. Most of these techniques have their origins in relational databases, and have been adapted to the monitoring domain.

In order to validate the relational approach, most of the components of the monitor were implemented and instrumented. Measurements show that the monitor can generate and process several hundred events per second, while at the same time presenting the conceptual viewpoint of time-varying relations to the user.

Acknowledgements

It is a privilege and a pleasure to thank my committee for their contributions to this thesis. Bill Wulf can take a good part of the credit in what research and writing skills I have gained in my years at CMU. Anita Jones was a constant source of good ideas, good questions (equally important) and encouragement. In Joe Newcomer I tapped a wealth of strategies for coming up with just the right abstraction. Zary Segall integrated my research efforts with several others in the department, generating a synergistic whole greater than the sum of its parts. Art Evans provided a valuable outside viewpoint, and did so under severe time constraints. Collectively, they represent almost 50 years of experience in designing, programming, and instrumenting distributed systems. This experience is evident in the ideas they generate and the systems they design and build. One would be hard pressed to find five people more suited to participate in this research.

In addition to my committee, other members of the department contributed to the thesis. The StarOS group, especially Bob Chansler and Ivor Durham, gave freely of their time in explaining the details of StarOS and modifying the operating system to test out my ideas. The other multiprocessor evaluation seminar participants, including Xavier Castillo and Ajay Singh, provided a forum for developing concepts in a friendly atmosphere. Peter Highnam helped me design the Ethernet protocol and implemented Medic, and Ivor Durham implemented the first version of the StarMon sensors. Karsten Schwans was helpful in all stages of the research. Barat Jayaraman, of the University of North Carolina, was instrumental in developing the TQuel semantics. M. Satyarayanan, Mark Sherman, and Steve Vegdahl were an immense help in the remote production of this document. Finally, Mike Accetta and Sharon Burks provided their unique expertise, administrative and otherwise, enabling me to finish in a reasonable amount of time.

Getting a Ph.D. is as much a matter of persistence as a matter of performance. My wife, Merrie Brucks, made the whole effort so much more enjoyable by her companionship, empathy, support, and affection.

To all, a sincere thank you.

Table of Contents

I. Approach

1. The Problem 3
1.1. The Cause and the Result 3
1.2. Definitions 3
1.3. The Impact of Complexity on Monitoring 6
1.4. Knowledge Representation 8
2. The Relational Model 11
2.1. Entities and Relationships 12
2.2. Time 13
2.3. Summary 14

II. A Temporal Query Language

3. An Informal Definition 19
3.1. The Quel Retrieve Statement 20
3.2. Adding Time to Quel 21
3.3. The TQuel Retrieve Statement 22
3.3.1. TQuel Expressions 23
3.3.2. Temporal Expressions 24
3.3.3. Event Expressions 26
3.4. The Temporal Selection Component 27
3.5. The Temporal Delimiter Component 27
3.6. Aggregate Operators 28
3.7. An Example 30
3.8. Summary 33
4. Semantics of the TQuel Retrieve Statement 35
4.1. Tuple Calculus 35
4.2. Path Expressions in TQuel 37
4.2.1. The Start, Stop, and At Clauses 38
4.2.2. The When Clause 40
4.3. Formal Semantics 44

4.4. Aggregate Operators 46
4.4.1. Informal Semantics 47
4.4.2. Formal Semantics 48
4.5. Indeterminacy 51
4.5.1. Semantics 52
4.5.2. Defaults 55
4.5.3. Indeterminacy and Aggregate Operators 57
4.6. Summary 59

III. Realization

5. A Low Level Data Collection Mechanism **63**
5.1. The Environment 64
5.2. The Mechanism 66
5.3. Integrating Sampling and Tracing 69
5.4. Other Uses for Receptacles 70
5.5. Interaction with the Remote Monitor 70
5.5.1. Naming 71
5.5.2. Time 74
5.6. Summary 75
6. The Update Network **77**
6.1. Incremental Updating of Temporal Relations 78
6.2. Generic Nodes 81
6.3. Access Nodes 81
6.4. Operator Nodes 82
6.5. Node Interconnection 82
7. Generating the Update Network **85**
7.1. Generating an Initial Network 85
7.1.1. Universal Relations 86
7.1.2. Compensation 88
7.1.3. Checkpoint Tuples 89
7.1.4. Other Details 90
7.2. Efficiency 91
7.2.1. Graph Transformation 93
7.2.2. Temporal Order 97
7.2.3. Limiting the Semantics of the When Clause 98
7.2.4. Node Scheduling 100
7.2.5. Other Issues 100
7.3. Summary 102

8. An Implementation 105
8.1. General Structure of the Monitor 105
8.2. Sensor Specification 108
8.2.1. Sensor Description Files 110
8.2.2. The Description File Preprocessor 112
8.3. The Resident Monitor 115
8.3.1. StarMon: General Structure 115
8.3.2. Sensor Performance Measurements 116
8.3.3. Medic 119
8.4. The Ethernet Protocol 119
8.5. The Remote Accountant 120
8.6. The Update Network 121
8.6.1. Tuple Representation 122
8.6.2. Interpretation and Compilation 123
8.6.3. Update Network Performance 124
8.6.4. Relationship to Data Flow 126
8.7. A Step Back 126
8.8. Evaluation 128

IV. Conclusion and Appendices

9. Conclusion 133
9.1. Surprises 135
9.2. Remaining Problems and Future Research 137
References 139
Appendix A. BNF of the TQuel Retrieve Statement 153
Appendix B. Proof of the Conversion Theorem 155
Appendix C. Operator Nodes 161
Appendix D. StarMon 167
D.1. The StarOS Task Force 167
D.2. The Monitor Object 167
D.3. Pipes 168
D.4. Receptacles and Sensors 170
D.5. A Microcoded Sensor 174
D.6. Efficiency 175
Appendix E. An Extended Example 177
E.1. Sensor Description File Processing 178
E.2. Description File Formats 179
E.3. Changes to the Source Code 181
E.4. Initializing the Monitor 183

E.5. Query Processing 186

Appendix F. Update Network Performance 191

F.1. The Unoptimized Version 192

F.2. The Optimized Version 195

F.3. The Compiled Version 196

F.4. Summary 200

List of Figures

Figure 2-1: Relationships between Primitive and Derived Events and Periods 15
Figure 3-1: Instantaneous versus Cumulative Count 30
Figure 3-2: Configuration of the PDE task force 32
Figure 4-1: The Overlap versus Coverage Interpretations for Combining Periods 40
Figure 4-2: Representing Uncertainty 53
Figure 4-3: Different Strategies for Handling Indeterminacy with Aggregate Operators 58
Figure 5-1: Skeletons of the Monitor and User Type Managers 67
Figure 6-1: Parse tree for the relational algebraic expression for the skeletal TQuel retrieve statement 80
Figure 7-1: An Initial Update Network 87
Figure 7-2: Monitor-Specific Transformations 94
Figure 7-3: Applying Multiple Transformations 96
Figure 8-1: Components of a Distributed Monitor 106
Figure 8-2: Components of the Monitor Core 108
Figure 8-3: Structure of the Monitor As Implemented 109
Figure 8-4: Sensor Description File Syntax 111
Figure 8-5: The position of DFPre in the program development process. 113
Figure D-1: Representation of the StarOS taskforce 168
Figure D-2: The Monitor Object 169
Figure D-3: The Structure of a StarOS Pipe 170
Figure D-4: Structure of an Event Record 171
Figure D-5: The Structure of a StarOS Receptacle 172
Figure D-6: The Parameter Block for the Sensor Instruction 175
Figure D-7: Sensor Description File for Measuring Sensors 176
Figure E-1: PDE Sensor Description File 178
Figure E-2: Dialogue with DFPre 180
Figure E-3: Definitions File Produced From PDE SDF 181
Figure E-4: The Description File Format (DFF) File for StarOS Sensor Descriptions 182
Figure E-5: Initialization Dialogue from Simon's Perspective 184
Figure E-6: Retrieving the Information Concerning the PDE 185
Figure E-7: PDE Query Contained in the file demoquery 186

Figure E-8: Parsing the Query 187
Figure E-9: Compiling and running the update Network 189
Figure E-10: Running the Update Network Using Generated Event Records 190
Figure F-1: Code for the Unoptimized Update Network as Generated by the TQuel Compiler 193
Figure F-2: The Unoptimized Update Network Generated by the TQuel Compiler, Part 1 194
Figure F-3: The Unoptimized Update Network Generated by the TQuel Compiler, Part 2 195
Figure F-4: The Hand-Optimized Update Network 197
Figure F-5: The Hand-Compiled Update Network 199

List of Tables

Table 8-1: Performance Measures for StarOS Sensors 117
Table 8-2: Performance Measures for the Update Network 125

I. Approach

The title of this dissertation has two components. The first component, Monitoring Distributed Systems, crisply states the problem. The second component, A Relational Approach, provides the solution. The central thesis of this work is that the relational model is an appropriate formalization of the information processed by a distributed monitor. The first part of this dissertation demarcates the problem and motivates the approach. The second part defines the language, first informally and then rigorously, used to query the monitor. Part III presents strategies for processing queries in this language, and examines an implementation of the monitor. Several appendices provide details that would blur the focus of the main text.

The two chapters in this part expand on the title. The first chapter will discuss in more detail what is involved in monitoring distributed systems, and why it is such an interesting and difficult problem. The second chapter introduces the relational model and lists the primary issues involved in the application of this model to monitoring.

Chapter 1
The Problem

1.1. The Cause and the Result

The cause is hidden, but the result is known.

-- *Ovid*, from Metamorphoses IV

In the realm of computing, as in all analytic endeavors, one must first understand the behavior before one can understand the underlying reasons for that behavior. As the computational structures employed in programs become complex, computer system designers, implementors, and users find it increasingly rare that they can agree with Ovid that "the cause is hidden, but [at least] the result is known." Monitoring is a necessary first step in understanding a computational process, for it provides an indication of *what* happened, thus serving as a prerequisite to ascertaining *why* it happened.

The realization that monitoring is a difficult task, one that deserves study, has come only recently to the computing community. When computing systems were simpler, it was possible to understand adequately the system's behavior with rather unsophisticated monitoring tools and (considerably more sophisticated) modeling techniques. Many aspects, such as characterizing the control flow or determining execution times, were so straightforward as to not even be considered issues. Times have changed, and many of these "non-issues" are now so problematic that present monitoring systems often do not provide *any* solutions to them.

1.2. Definitions

The definition of monitoring employed in this dissertation is a rather general one: monitoring is the extraction of dynamic information concerning a computational process, as that

process executes[1].

A *computational process* is anything that can be said to compute[2]. Examples include a microprogram, a subroutine, a conventional process, a collection of processes, or even an operating system. Computational processes vary in at least two ways concerning monitoring: (1) the number of components to be monitored, from a single wire on a bus to the entire system, and (2) the time frame in which the measurements take place, ranging from tens of nanoseconds to months. The level of abstraction (the *granularity*) at which the monitoring takes place has a substantial effect on the methods used to collect data.

Dynamic information may also be spread over a large range of temporal granularity, from information concerning the sequence of microinstructions executed during a particular time interval, to the average amount of time a routine executes, to some global statistic concerning the execution of a whole series of programs. If the information to be collected is not dynamic, there is no need to collect it as the process executes.

Defining a distributed system is difficult. Although John Shoch has several arguments to support the contention that "there is nothing different about 'distributed' computing" [Shoch 81], he also presents several distinctions between distributed and non-distributed systems (his widely-shared belief is that there *is* a difference). The two relevant to monitoring are

- Distributed systems are characterized by a lack of central control.

- A quantitative difference in the number of system components (processors, memory, addressing domains, etc.) leads to a qualitative difference[3].

---

[1] There are at least two other definitions of *monitor* which should be mentioned. One use of the word monitor, prevalent in the 1960's and early 1970's, is as a synonym for *operating system* or at least the user interface of an operating system. The second refers to an arbiter of access to a data structure in order to ensure specified invariants, usually relating to synchronization [Hoare 74]. Both definitions emphasize the *control*, rather than the *observational*, aspects of monitoring. The term monitor as used in this dissertation is the (usually software) agent performing the monitoring.

[2] *Italics* will be used for introducing new terms and for emphasis.
Boldface will be used for reserved words.
SMALL CAPITALS will be used for names of relations.

[3] At the same workshop [Liskov 81], Richard Watson added several related attributes: more heterogeneity, distribution of state, and communication via messages. David Reed offered perhaps the best argument for monitoring in his characterization of distributed systems: "In centralized systems, it has been possible so far for single persons to understand the entire system (even of the size of MULTICS). This will not be possible for distributed systems. How can we comprehend parts of the system without comprehending all of it?" Two other characterizations of distributed systems have been proposed which will be quite important in monitoring: (a) a completely accurate global clock is not available, and (b) once a remote action has been requested, the requester cannot always determine, in a bounded time, whether or not it has occurred. [Enslow 78] provides yet another definition.

These two aspects conspire to make monitoring a distributed system a difficult (and thus interesting) task. A precise definition of 'distributed' is not important; the intent of the title is to include the above attributes in the problem domain.

The general definitions presented above allow concepts developed in this research to be applied to several previously unrelated domains. This section closes with a discussion of representative utilizations of monitoring information.

One use of monitoring is to facilitate the debugging of complex programs. Debugging proceeds in five stages [Model 79][4]: (1) observe the behavior of a computer program; (2) compare this behavior with the desired behavior; (3) analyze the differences; (4) devise changes to the program to make its behavior conform more closely to the desired behavior; and (5) alter the program in accordance with these changes. Monitoring is concerned with the first two stages in this process. The third and fourth stages are still the province of the programmer (although the Programmer's Apprentice project [Shrobe 79] is making progress in this area); the fifth stage is routinely accomplished using text editors, and could be automated given the automation of the fourth step.

A second use of monitoring tools is in making efficient use of limited computing resources. Ideally, optimization of resources would be done analytically, but in general *a priori* determination of runtime efficiency is impossible. Thus it is necessary to tune the application once it is implemented. Tuning requires feedback on the program's efficiency, which is determined from measurements on the application while it is running[5].

A third use of monitoring is to query the system, not for performance measures, but merely for status information, such as how far a computation has progressed, who is logged on the system (the *system status* command of most time-sharing systems), the state of certain files (the *catalogue* or *directory* commands), or the quantity and nature of hardware and software failures.

And finally, monitoring information may also be used internally by the application program for various purposes. For example, consider a program which varies the number of processes dedicated to a particular function based on the request rate for that function. Information concerning the hardware utilization and the number of outstanding requests could be used by the program to determine whether to start up more processes to handle the current demand (if the utilization is low and the request rate high) [Rashid 80a, Wulf *et al.* 75a]. Monitoring information is also valuable for programs which must be reliable; the fact

---

[4] Joseph Newcomer points out that this process is essentially the scientific method.

[5] This tuning has been termed *performance debugging*: it's not enough just to show that a system works; you want it to work well [Liskov 81].

that a processor (containing particular processes belonging to a program) has failed, for example, is important to the program if it must be able to recover from such failures[6].

1.3. The Impact of Complexity on Monitoring

The previous section indicted that monitoring is difficult because of the complexity and decentralization of the process being monitored. The purpose of this section is to determine how increased complexity impinges on the task of monitoring. The impact of decentralization is reflected more in the specific algorithms and will be dealt with in later chapters. We will start by investigating the monitoring of the program counter, certainly an important aspect of the dynamic state.

In the "good old days", a program consisted of a single program executing on a monolithic operating system on a single processor. The program counter could be traced or sampled. Tracing, which involves storing the information each time some event occurs, is usually done at the procedure, statement or individual instruction level, with a concomitant increase in overhead. At the beginning of each procedure (or statement or instruction), code is inserted by a preprocessor to increment a counter or generate a timestamp. A postprocessor is often used to correlate the data with the source text of the program.

Sampling involves storing information asynchronously with the execution of the program. Usually sampling is initiated by a clock tick, by an operating system call, or by a separate process. The information gathered by sampling is stochastic; for instance, it can indicate what percentage of execution time takes place within an individual routine, but it cannot reliably determine how many times a routine was invoked. Sampling has the advantage that it requires fewer resources, and thus perturbs the system to a smaller degree than tracing.

In the past two decades the programming environment has changed radically. In some sophisticated systems being developed today, a program consists of many interacting processes running on many geographically distributed computers communicating over high bandwidth networks [Clark 78]. These systems differ quantitatively with systems of the past: where there was one processor, there are now tens to hundreds; where there was one process, there are now many per processor; where there were a few I/O devices, there are now complex communication media, sophisticated encoding formats, and powerful inter-process communication protocols, all supported by large software components; where there was a single contiguous address space, there are now many small, separately addressable objects, each containing code or a specific data structure.

---

[6] Eric Rosen, in an article describing a particularly interesting instability which occurred on the ARPANET, concluded that "we need a better means of detecting that some high priority process in the Imp [a node on the ARPANET], despite all the safeguards we put in, is still consuming too many resources." [Rosen 81]

Returning to the example of monitoring the program counter, we must first determine what the "program counter" means in a distributed system. One possibility is to use the program counter of each of the processes making up the program of interest. For the single process example, the routine name and statement number within that routine may be quite informative; a printout of, say, fifty routine names and statement numbers is rather overwhelming. This *quantitative* difference necessitates a *qualitative* change in the monitor, for there is one aspect that remains unchanged: the user (and especially the information capacity of the user) must still interpret the monitoring data.

The presence of the user has been implicit throughout this discussion. Fundamentally, the user is not interested in the program counter at all; instead, the user wants an understanding of the *state* of the execution as it evolves through time. This state manifests itself in many forms: the changing values of the variables in the program, the input read by the program and the output produced, the constantly changing program counter. All are valid components of the program state, and each may be sufficient when monitoring a single process. Individually, and in their raw form, however, they are woefully inadequate for monitoring distributed systems, because there is simply too much information, most of it irrelevant. Instead, the monitor must be able to express the system state (as well as other attributes of the system) in a form useful to the user.

As an example, suppose the monitor could provide this description of the program state:

> Process A is waiting on process B to acknowledge the xxx request; process Y is sending process Z information concerning the object yyy; and process M has completed.

There are several aspects to note in this example. The information the monitor displayed is both less and more than a list of program counter's. The monitor had to understand that a program counter in a certain range meant that process A was waiting for something, yet the exact program counter was unimportant. Conceivably, the program counter could have been completely different and the monitor would have displayed the same information. In addition, the monitor had to be able to look inside the various queues and buffers maintained by the communication mechanism in order to be able to state that a process is waiting on another process to acknowledge a particular request. Names had to be associated with the various processes, objects, and requests in order to produce an intelligible state description. And finally, the monitor had to know that the user was interested in the current state in terms of interprocess communication. Another perhaps just as useful state description is

> Process A has used 75% of its resources, while processes X, Y and Z have used only 20% of their resources.

The decentralization inherent in distributed systems also necessitates interpretation of the monitoring data. The mention of several processes in the previous example implies a degree

of logical decentralization; if those processes are on different processors, then there is also physical decentralization. To present a global view of the program state, the monitor must integrate data collected at geographically distinct sites. Simply determining what information to collect and where to acquire this information becomes a difficult task. Hence the quantitative and decentralized aspects of the monitored system, coupled with the limitied information handling capabilities of the user, demand an intelligent monitor. The next section will discuss the organizing concepts for a monitor which can collect information from a variety of sources, interpret this information, and present it in a series of high level views in a format comprehensible to the user.

1.4. Knowledge Representation

In its most general form, the process of monitoring is concerned with retrieving information from the monitored system and presenting this information in a derived form to the user. Viewing the monitor as the proverbial black box, it is fundamentally an information processing agent. As the previous sections have indicated, this activity is rather sophisticated. Looking inside this black box, there is some form of knowledge representation to direct the monitoring activity. Thus, there are at least two ways to view a monitor abstractly: as a knowledge representation system and as an information processing agent. As will be seen, both of these views are fruitful. The rest of this chapter will investigate the knowledge representation issues; chapter 2 will pursue the information processing aspects of monitoring.

In an examination of the discussion of a possible high-level program counter, one starts to notice phrases such as "the monitor had to understand." In one sense, the monitor *can't* understand; it is, after all, only a computer program. However, computer programs are remarkably versatile (c.f. Church's Thesis) and almost any type of desirable behavior can be programmed with the correct selection of data structures and algorithms. Hence, the process of "teaching the monitor" or "making the monitor understand" is transformed into the more intellectually manageable task of deciding what data structures and algorithms to employ within the monitor.

These data structures and algorithms encode the knowledge the monitor can apply to the task at hand. Existing monitors perform little interpretation of the collected data, and thus use rather *ad hoc* methods for determining what to monitor and how to perform the monitoring. Two recent systems have addressed the monitoring of complex systems; it is useful to analyze the character of knowledge each used to direct the data collection and interpretation.

Model's thesis [Model 79], one of the first to approach this topic systematically, stressed the adoption of a uniform model of a complex activity for use in monitoring. His monitor was designed to be used with programs implemented in artificial intelligence languages such as KRL, which are themselves implemented in Lisp. Despite the sophisticated control and data

structures provided by these high level languages, most debugging is still done in the implementation language. The complexity of programs written in these languages is seriously limited by the lack of adequate debugging tools. Model argued that of the five stages present in the debugging process (see section 1.2), monitoring has the most potential for improvement at this time.

The monitor collected events generated by the interpreter (the monitor had no control over which events were collected). These events were related to the program's data and control structures implicitly in the routines generating the events. However, some cross-referencing was done, so that the monitor knew, for example, that some events caused other events. The user could specify which events, as well as which fields in these events, were to be displayed. The knowledge utilized by the monitor was wired into its code.

Gertner's thesis [Gertner 80] focussed on the flow of messages between processes in RIG [Ball *et al.* 76], a distributed system constructed at the University of Rochester. In his system, Gertner described the computation using finite state automata, with the transitions being events (usually messages sent between two processes in the system). Associated with each message is a set of timestamps relating to the activity involved in processing the message. These timestamps allow the monitor to calculate processing intervals, message counts, overlapping periods, etc. A hierarchy of finite state automata can be defined, with elementary transitions at one level composed of multiple transitions at a lower level. This hierarchy allows monitoring information to be presented at the appropriate level of abstraction. Again, the knowledge of how to derive information from the timestamps was implicit in the monitor's code.

Unfortunately, these approaches are simply inadequate for distributed systems. In Model's system, events capture only the notion of *state transitions*. The system state must be inferred by the user. Modeling all activity in terms of finite state automata, as in Gertner's system, while expressing to some degree the semantics of the periods between the events, is overly restrictive. Sampling data (as opposed to trace data) does not integrate easily into the scheme. The proliferation of extraneous states is also a problem which results from a total reliance on this model. Because of the restricted modules built into these systems, it is unclear how the systems could be extended to eliminate these problems.

In order to construct a monitor which can apply substantial knowledge concerning the system being monitored, this knowledge must be organized in a coherent fashion. Thus a formalism is needed to describe this knowledge. The formalism must, to some degree, encode the following knowledge:

- what information the monitor collects concerning the system;

- how new information can be acquired by the system;

- what dependencies exist between various components of this information;

- how the information relates to the data and control structures within the programs, and to the data and control structures of the underlying operating system; and

- what information the user wants to see.

There are also three basic notions that must be characterized by this formalism: entity, relationship, and time. The monitor must understand that there are such things as processors, processes, memory, message ports, semaphores, etc. and that certain relationships exist between these things, such as a process running on a processor. In Model's thesis, for example, entities were the values of certain attributes, and the relationships were the events themselves.

The third notion is that of time. The monitor must understand that facts are only true for a certain period of time, and that entities and relationships are temporally bounded. For instance, in Model's thesis, time was one of the fields in each event record, and queries could specify which time period the user was interested in. Also, Model's monitor understood that events were sequential, and thus that some events were after others. However, the concept of something being true for a period of time *between* two events is not represented within the monitor, and thus the user could not request such information. Clearly, a multiprocess monitor must have a better understanding of time.

The aim of this chapter has been to sufficiently refine the original problem statement into one which can be attacked in concrete terms. This chapter has argued that monitoring is concerned with knowledge representation and information processing. The next chapter will investigate the information processing aspects of monitoring.

Chapter 2
The Relational Model

Viewed abstractly, a monitor collects, manipulates, stores, and displays information concerning the dynamic state of a computational process. It is fundamentally an information processing agent: the information describes temporal relationships between entities involved in the computation, and the processing is quite sophisticated, due to the cognitive limitations of the user. Previous work on monitoring has concentrated on techniques for collecting monitoring data. As the previous chapter demonstrated, such a view is inadequate. The approach taken in this thesis is the information must be structured in such a way that manipulating it is straightforward. Also, there must be powerful algorithms which can satisfy highly variable requests from the user.

A great deal of research has considered effective ways to process information. One of the results of this research has been the *relational model* [Codd 70]. The relational model provides both a structuring of the information and operations on that information. Information is stored in *relations*. A relation models a particular relationship between collections of entities. Relations can be thought of as tables having a number of rows and columns. The rows are called *tuples* and the columns *domains*. Each tuple of a relation models a particular relationship between entities named in the domains of the tuple. For example, the relation

```
Employee (Name, Salary, Department)
```

might include the tuple (Huttinger, 44000, Commerce). New relations can be derived from existing ones, using one of several data manipulation languages developed for the relational model; these *query languages* are syntactically simple yet are remarkably powerful in their expressiveness. One important aspect of some query languages is that they are declarative rather than procedural: they specify *what* information is desired, rather than *how* this information is to be derived. One possible query on the Employee relation would be to retrieve into a relation GivePerks (Name) all the employees making more than some minimum salary.

The central thesis of this work is that the relational model is an appropriate formalization of the information processed by the monitor. The primary benefits include a simple, consistent structure for the information and the existence of powerful declarative query languages. Previous uses of the model have been confined to static databases. The remainder of this chapter discusses how the relational model is applied to the monitoring domain.

2.1. Entities and Relationships

In order to use the relational model in monitoring, monitoring analogues must be specified for each of the components of the model. We will be more formal in order to characterize the application of the relational model to the monitoring domain precisely.

A relation is any subset of the Cartesian product of one or more domains [Ullman 82]. A domain is simply a set of values. These values may be *literal*, such as integers or character strings, or they may be names of *entities*. Entities are conceptual objects which exist independently within the system being monitored. Entities may have a physical realization, such as a processor, a disk, a line on a bus, or a word in memory. Alternatively, entities may be virtual, such as a user job, an activity queue, or a capability list. Entities have *names* (both internal and user-oriented) which allow them to be identified.

There are two types of relations in the monitor: *primitive* and *derived* relations. Both represent relationships between entities. Each tuple of a relation indicates a particular relationship between the entities named in the domains of the tuple. An example is the RUNNINGON relation, which has two domains: a process and a processor. The tuple (Process17, Processor5) in this relation represents a fact concerning Process17 and Processor5, namely, that Process17 is running on Processor5.

Conceptually, each primitive relation is associated with a predicate that is true if the relationship is satisfied for a given set of entities. This association provides a well-defined semantics for the relation, since a particular tuple is in the relation if and only if the predicate returns true when applied to that tuple. The predicate for the RUNNINGON relation might be "the process is in the run-queue of the processor." Primitive relations may exist at any monitoring granularity in the system. The sole requirement is the ability to specify the predicate as some function of the system state accessible to the monitor.

Primitive relations are often just that. The user is probably not interested in the level of detail present in the primitive relations; instead, the user desires more summary information extracted from this detail. Query languages provide a powerful mechanism for specifying exactly the information the user wants to retrieve from the monitor concerning the system. In this way, information not anticipated by the designer of the monitor is still available to the user, provided the basic information is available to the monitor through the defined predicates. An example of a derived relation is RUNNINGONPROCESSOR5, containing a Process domain, which would contain exactly one tuple at any instant of time, the process which is currently running on Processor5.

Section 1.4 listed three notions to be characterized: entity, relationship, and time. The first two have been are modeled directly in the relational model. The third is perhaps the most fundamental, for without the system state changing as time progresses, there is no need to

monitor the system. Hence, it is important that the notion of time be consistently represented within the monitor.

2.2. Time

Time goes, you say? Ah no!
Alas, Time stays, we *go.*

-- *Austin Dobson*, in "The Paradox of Time"

Time by itself does not exist; but from things themselves there exists a sense of what has already taken place, what is now going on, and what is to ensue. It must not be claimed that anyone can sense time by itself apart from the movement of things or their restful immobility.

--*Lucretius*, in De rerum natura, Book I

Within the monitor, relations are differentiated temporally: there are *event* relations and *period* relations. A tuple in a period relation specifies a relationship valid during the time interval $[t_0, t_1]$. The RUNNINGON relation described earlier is a period relation, since each particular tuple is true for a finite stretch of time. Since the relation is a collection of tuples, it is also a collection of periods.

A tuple in an event relation describes a change in the state of the system which occurred at a particular instant of time. Events delimit periods: a given event causes one or more periods to start; other event(s) cause the period(s) to stop. An example is the STARTRUNNING event relation, with processor and process domains. The tuple (Process17, Processor5) in this relation represents the instantaneous event of Process17 starting to run on Processor5. This event caused a similar tuple in the RUNNINGON period relation to be true.

Event and period relations are *complete*, in that a succession of system states at a particular level of abstraction (the *monitoring granularity*) can be determined by the appropriate event or period relations. For example, either the RUNNINGON or the STARTRUNNING relations are sufficient to specify when each process was running on each processor. Since they are complete, they are also *duals*, in that the tuples in a period relation can be determined given the appropriate event relations, and an event relation can be generated given the appropriate period relations. Hence, knowing which events occurred provides the periods which were started or stopped by those events, and knowing when the periods started and stopped provides the events that occurred. Although this duality depends on a number of assumptions which are often difficult to satisfy in practice, it is important because it provides a way to accommodate both *sampling* data, related to period relations, in that successive samples provide the successive tuples of a period relation and *tracing* data, related to event relations, which are generated by a stream of events.

Because relations can be derived from other relations and relations represent events and periods, it is possible to specify derived events and periods. The query language must be augmented with additional semantics to permit the specification of such temporal relationships as simultaneity and consecutivity. Figure 2-1 illustrates how derivation and duality interact.

2.3. Summary

This chapter has provided the fundamental thesis for this research:

> The information collected by the monitor should be presented to the user as a collection of time-varying relations which can be manipulated by a temporal query language.

This contention was supported by considerations of programming environment complexity, cognitive limitations of the user, and the underlying functionality of the monitor. There remains one problem: how can the relational view as presented to the user be supported effectively by the monitor? In this context, effectiveness implies powerful, user-friendly, efficient and system-independent, all interrelated attributes. Relations are structurally simple, and the query languages are straightforward. The concept of a dynamic database of information on the behavior of the system is easy to comprehend and use. The user still specifies *what* information is desired; the monitor must apply all of its knowledge to determine *how* to supply this information: what information to collect and what manipulations on this information are necessary, and it must do this in an efficient manner. The goal is to construct a monitor which can support the relational model in its full generality for the user, yet perform the actual monitoring as effectively as a manually constructed monitor tailored to the specific task.

This dissertation is loosely organized around a sequence of problem and result statements. Implicit in each problem statement are the results generated by preceding statements, because one benefit of acquired knowledge is the ability to ask further, more precise questions. Given the framework presented so far, it is possible to ask several general questions:

> **Problem:** How may traditional query languages be extended for use in monitoring?
>
> **Problem:** Is it possible to provide effective data collection mechanisms?
>
> **Problem:** How can the dynamic incremental updating of temporal relations be implemented effectively?
>
> **Problem:** How can knowledge be used to direct the processing of user queries?

The following chapters will present solutions to all of these problems.

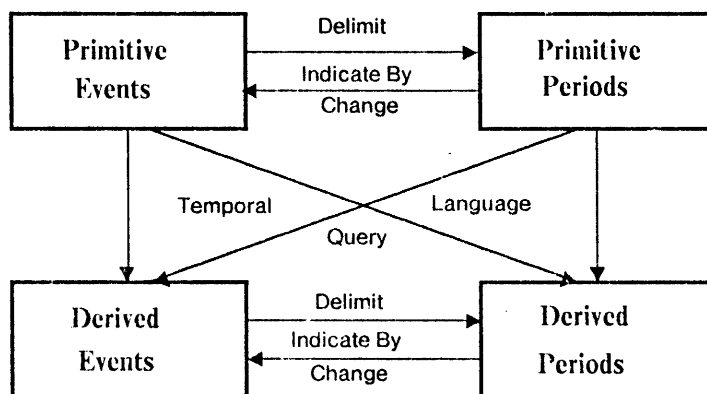

Figure 2-1: Relationships between Primitive and Derived Events and Periods

II. A Temporal Query Language

The two chapters in this part define the language used to query the monitoring data base. An informal definition is given first, emphasizing the way in which the constructs of the language may be used to make meaningful queries involving time. The second chapter then provides a formal semantics for this language. In a first reading, the reader is encouraged to skip most of chapter 4, reading only the introduction to the tuple calculus and the summary (sections 4.1 and 4.6).

Chapter 3
An Informal Definition

This chapter describes a language for querying a temporal data base such as the one supported by the monitor. The user indirectly specifies the actions to be taken by the monitor by describing the desired information in this language. As a result of processing the query, the correct information is collected, processed, and presented to the user. There are two prerequisites for this scenario to be realized: the user's query must contain a complete specification of the desired information (with as little irrelevant detail as possible), and the monitor must be able to take this abstract specification and "do the right things" with it. This chapter will deal with the specification itself, the query language and the next chapter will present a formal semantics for the language.

The language is a strict superset of Quel [Held *et al.* 75], called *Temporal QUEry Language,* or TQuel. Quel, used in the Ingres relational database system [Stonebraker *et al.* 76], is a relational tuple calculus language with 16 basic statement types: **Append**, **Copy**, **Create**, **Define**, **Delete**, **Destroy**, **Help**, **Index**, **Integrity**, **Modify**, **Permit**, **Print**, **Range**, **Retrieve**, **Save**, **View**. These statement types, presented in detail in [Held *et al.* 75], support the creation and destruction of databases and relations, storage structure modification, bulk copy of data, consistency, integrity, and concurrency control, retrieval of information, and miscellaneous operations. Since the task of monitoring concerns primarily the retrieval of information, only changes to the **retrieve** statement were investigated. A full implementation of a monitor would also have to make modifications to several of the other statement types. Since TQuel is a strict superset of Quel, every acceptable Quel **retrieve** statement is also an acceptable TQuel statement. The complete syntax for the TQuel **retrieve** statement is given in Appendix A.

TQuel augments the **retrieve** statement with additional syntax, and provides a more comprehensive semantics by treating time as an integral part of the database. Before investigating these changes, it is instructive to examine the original Quel **retrieve** statement. The Quel examples all concern the following relation:

```
Employee (Name, Dept, Salary, Manager, Age)
```

3.1. The Quel Retrieve Statement

Informally, the Quel **retrieve** statement selects a subset of the tuples in one or more relations, extracts one or more domains from the tuples in this subset, and combines the domains into result tuples. The **retrieve** statement works in conjunction with the **range** statement (the syntax is presented in standard BNF, with ε designating the empty string and nonterminals in brackets):

```
<range statement> ::= range of <tuple variable> is <relation>
<tuple variable> ::= <name>
<relation> ::= <name>
```

The statement

```
range of E is Employee
```

specifies that the <tuple variable> E will be represent the tuples of the relation EMPLOYEE on any subsequent **retrieve** statements, until E is redefined by another **range** statement. The expressions appearing in the **retrieve** statement contain constants and previously defined <tuple variable>s.

The **retrieve** statement creates a new relation (perhaps only temporarily, if no <relation> is provided) with the domains named in the <target list> whose tuples satisfy the <boolean expression>. For example, to compute the salary divided by Age - 18 for each employee in the Toy department:

```
retrieve into T (Comp = E.Salary / (E.Age - 18))
where E.Dept = "Toy"
```

results in a new relation T which has a single domain Comp calculated for each qualifying tuple. The <target list>

```
(Comp = E.Salary / (E.Age - 18))
```

specifies the domains of the new relation. In this case, T will contain one domain called Comp, computed from the Salary and Age domains of the tuples in EMPLOYEE. The <where clause>

```
where E.Dept = "Toy"
```

specifies which tuples will contribute toward the new relation. The **retrieve** statement thus consists of a *domain specification* component (the <target list>) and a *tuple selection* component (the <where clause>). Each may be defaulted; the <target list> to all the domains in one of the underlying relations; and the <where clause> to

```
where true
```

The complete syntax is as follows:

```
<retrieve statement> ::= <retrieve head> <where clause>
<retrieve head>      ::= retrieve <into> <target list>
<into>               ::= ε | unique | <relation> | into <relation>
<target list>        ::= ( <tuple variable>. all ) | ( <t-list> )
<t-list>             ::= <t-elem> | <t-list> , <t-elem>
<t-elem>             ::= <name> <is> <expression>
<is>                 ::= is | =
<where clause>       ::= ε | where <Boolean expression>
<relation>           ::= <name>
```

3.2. Adding Time to Quel

While ladies draw their stockings on,
The ladies they were are up and gone.

--*Ogden Nash*, in "Time Marches On"

Since TQuel must be able to express queries on temporal relations, the **retrieve** statement was augmented with additional semantics involving time. As introduced in chapter 2, temporal relations are collections of tuples, each representing either an event or a period. Hence, each <tuple variable> appearing in a **retrieve** statement will constitute both an assignment of values to domains, and a time element. The **retrieve** statement thus becomes a way to specify the combination of one or more periods and/or events into a resulting period or event. This characterization, as we will see, has far-reaching consequences on the syntax, semantics, and processing of the TQuel **retrieve** statement.

There is a fundamental decision to be made when adding time: should the time domain be explicit, that is, directly manipulatable by the user in the <target list> and <where clause>, or implicit, manipulatable only through additional clauses in the language. TQuel adopts the latter approach, for several reasons.

One reason TQuel provides separate clauses for the time domain is that the alternative, allowing the user to manipulate time as simply another domain, attempts to ignore an important aspect of the database. Time is fundamental in a temporal database and significantly impacts the processing of such a database (this aspect will be considered in detail in subsequent chapters). Ultimately, the fact that a tuple has a domain specifying the time it was valid determines to a major extent the processing of that tuple. To the database management system (i.e., the monitor), the time domain is much more important than the other domains found in the tuple.

A second reason involves the operations allowed on the proposed time domain. The statement (in an imaginary query language)

```
retrieve . . . . .
where A.time < B.time
```

implicitly states that the tuple represented by the <tuple variable> A was generated *before* the tuple represented by B. Not only must the monitor discover this fact, but so must the user. A perhaps more satisfying syntax is

```
retrieve . . . . .
when A before B
```

The semantics in the second case is clearer both to the monitor and the user.

Finally, the monitor should permit as much flexibility as possible, without enforcing inane rules to disallow semantically incorrect queries. For example, the query

```
retrieve (D = A.D, Time = A.Time)
where B.Time < (A.Time or C.Time)
```

might have the semantics that B must be followed by A or C, and that the time of the result tuple is to be the same as the underlying tuple, represented by the <tuple variable> A. However, another, quite similar, query:

```
retrieve (D = A.D, Time = A.Time or C.Time)
```

does not make sense, although the types still match. The problem is that the time domain is being used in several different ways, yet the same syntax is being applied in all cases. In TQuel, the first statement (in an appropriate syntax) is allowed, while the second is not.

3.3. The TQuel Retrieve Statement

For the reasons expressed above, the approach taken with TQuel was to make the time domain an implicit one, and to extend the **retrieve** statement with clauses dealing with this implicit domain. As discussed earlier, the Quel **retrieve** statement consists of a domain specification component and a tuple selection component. TQuel augments the statement with two analogous components: the *temporal delimiter component* and the *temporal selection component:*

```
<retrieve statement> ::= <retrieve head> <retrieve tail>
<retrieve tail>      ::= <selection> <temporal delimiter>
<selection>          ::= <where clause> <temporal selection>
```

The only other change is that the <target list> may be empty, specifying an event or period relation with no explicit domains.

In TQuel, each relation represents a collection of events or periods; for simplicity, let us restrict the discussion to periods. Each tuple (period) of the resulting relation consists of domains from the tuples (periods) of the underlying <tuple variable>s. The combinations of the underlying tuples which are accessible is determined by the selection component (on the explicit domains) and the temporal selection component (on the implicit time domains). The domains of the result tuple are determined from the explicit domains by the domain specifica-

tion component and the implicit time domain from the time domains of the underlying tuples by the temporal delimiter component. Thus, for each component of the **retrieve** statement concerning the explicit domains, there is an analogous component for the implicit time domain.

Before going into the details, an example is appropriate. The following relation will be used:

```
RunningOn (Process, Processor)
```

RUNNINGON is a period relation with two domains: a processor and a process. Each tuple in this relation describes a period of time when a particular process was running on a particular processor. The query[7]:

```
range of R is RunningOn
retrieve StartRunning (R.Process)
where R.Processor = Processor1
when "3:00pm" ; R
at R.start
```

derives an event relation called STARTRUNNING with one domain: a process. Each tuple in this relation describes an instant of time when a particular process started running on Processor1. If the processor was multiplexed among many processes, there could be many tuples in STARTRUNNING, one for each time the processes was restarted. Only events occurring after 3:00pm are included in this relation (the expression α ; β specifies that α precedes β[8]). In this query, the selection component consists of the <where clause> <when clause>, and the temporal delimiter component consists of the <at clause>.

3.3.1. TQuel Expressions

Before we discuss the additional components, we should consider the format of expressions. Standard Quel domains may be of one of the following types: fixed length character string, integer (1, 2, and 4 bytes long), and floating point (4 and 8 bytes long). TQuel augments these domain types with a *temporal* type, whose value is a linear function of time. A temporal domain is initially created using the *duration* function, which requires a <tuple variable> as an argument, and whose value is the length of time the tuple is valid. A temporal domain may be operated on by any arithmetic operator, as long as the result is either one of the standard types or is itself a ratio of two linear functions of time (see section 6.3).

Quel allows the standard arithmetic, string, trigonometric, and type conversion operators to be performed on domains. Quel also includes a few *aggregate* operators (count, average,

---

[7] Short examples of the various components of the TQuel **retrieve** statement will appear throughout this chapter. A complete example may be found in section 3.7.

[8] This somewhat unusual syntax was taken from path expressions (see section 3.3.2). See [Shaw 80] for a review of non-procedural notations based on regular expressions.

sum, minimum, and maximum) which return the same value for a collection (aggregate) of tuples. TQuel augments these operators with temporal semantics (see section 3.6).

One way to apply operations to the implicit time domain is to redefine the semantics of the operators defined on the explicit domains. For example, addition might mean "later", subtraction "earlier", and equality "at the same time". This approach may be characterized as arbitrarily associating syntax (algebraic operators) and semantics (later, earlier, etc.). This approach was not adopted in TQuel because it encourages quite confusing constructions, and also suffers from most of the drawbacks mentioned earlier of making time an explicit domain.

The approach taken in TQuel was to define three types of expressions: standard expressions, <temporal expression>s, and <event expression>s. <temporal expression>s evaluate to a Boolean value indicating whether the ordering specified by the expression was satisfied. The clause

```
when "3:00pm" ; R
```

includes a <temporal expression> specifying that two events be sequenced in time. <event expression>s evaluate to a timestamp indicating a particular event. The clause

```
at R.start
```

contains a particularly simple <event expression>. The use of <temporal expression> and <event expression> in the TQuel **retrieve** statement will be discussed after examining the syntax and informal semantics of these expressions.

3.3.2. Temporal Expressions

There are several straightforward examples of <temporal expression>s: a <tuple variable>, the ticking of a clock, a particular clock time. Since events can be derived from periods (see section 2-1), the start and stop events of a period (the start and stop events of an event are simply the event itself) are included, as are string and integer constants. The string constant specifies a wall clock time (such as "3:00pm"); and the integer specifies the number of time units (such as milliseconds) since the start of the session with the monitor. Both are examples of temporal constants. The ticking of a clock is accommodated by a predefined event relation CLOCK, which contains a tuple designating every "tick".

The above <temporal expression>s all define events, which do not specify interesting orderings. Any nontrivial <temporal expression> must be composed of more than one event or period. The most straightforward is a sequence of two events, defining a simple ordering of the events. TQuel allows more general expressions: an ordering may be specified as a regular expression on the participating tuples (e.g., the <tuple variable>s). The syntax is that of *path expressions,* which are regular expressions augmented with parallel operators [Habermann 75, Andler 79]. Path expressions, as originally defined, specify *constraints* on

the dynamic behavior of the program. In TQuel, path expressions are used in two separate ways in <temporal expression> and <event expression>. Indeed, the original motivation does not apply, since the operations have been performed and the event records generated before the processing of those records commences. Path expressions are used in <temporal expression>s to specify the relative ordering of the tuples participating in the query. The <temporal expression>

```
"3:00pm" ; R
```

appeared in the above example. The other application of path expressions is in <event expression>s, where they are used to select an event. <event expression>s, such as R.start appearing above, will be discussed later.

The following is the syntax for <temporal expression>s:

```
<t-exp>      ::= <element>
             | <t-exp> , <t-exp>
             | <t-exp> ; <t-exp>
             | <t-exp> "|" <t-exp>
             | <t-exp> . time
             | <t-exp> . start
             | <t-exp> . stop
             | ( <t-exp> )
<element>    ::= <tuple variable>
             | <string>
             | <integer>
```

The syntax of ". start" and ". stop" is designed to exploit the user's mental image of accessing the implicit time domain of the result of the expression (sing a syntax reminiscent of record accessing). The same observation holds true for ". time". The informal semantics are

.start	indicates the starting event;
.stop	indicates the stopping event;
$\alpha ; \beta$	specifies that β must follow α in sequence;
$\alpha \mid \beta$	specifies that at least one of the two expressions must be true; and
α , β	specifies that the two executions must overlap in time.

Examples will appear shortly.

3.3.3. Event Expressions

<event expression>s are quite similar to <temporal expression>s. The major difference is the selection operator, which is found in <temporal expression>s but not in <event expression>s. The syntax is therefore almost identical to that of <temporal expression>s:

```
<event expression> ::= <element>
                     | <event expression> . time
                     | <event expression> . start
                     | <event expression> . stop
                     | <event expression> ; <event expression>
                     | <event expression> , <event expression>
                     | ( <event expression> )
```

The informal semantics of <event expression>s are

.start — selects the start event;

.stop — selects the stop event;

$\alpha ; \beta$ — forms a period starting when α starts and stopping when β stops; and

α , β — forms a period starting when the second period starts and stopping when the first period stops, thereby determining the interval of time when both α and β were valid.

To illustrate the difference between the two types of expressions, the <temporal expression>

```
(A ; B)
```

may be used to specify the condition that the tuple associated with A occurred before that associated with B. To select the tuple which occurred first, the following <event expression> would be used:

```
(A , B).start
```

This expression specifies that the tuples (events) associated with A and B occur in parallel, and that we are interested in the one occurring first.

3.4. The Temporal Selection Component

The <temporal expression> and <event expression> are used in the additional constructs of the TQuel **retrieve** statement. The temporal selection component, the temporal analogue to the <where clause>, specifies the desired temporal ordering of underlying tuples participating in the derivation:

```
<temporal selection>    := ε | when <tbool-exp>
<tbool-exp>             ::= <t-exp>
                        | ( <tbool-exp> )
                        | <tbool-exp> and <tbool-exp>
                        | <tbool-exp> or <tbool-exp>
                        | not <tbool-exp>
```

The <when clause> selects tuples based on their ordering in time, rather than on the values of their domains, as the <where clause> does. The <when clause> includes a logical expression, which in turn contains <temporal expression>s. It is satisfied if the tuples associated with the <tuple variable>s found in the clause do in fact satisfy the <temporal expression>. For example,

```
when A ; (B | C)
```

specifies that the tuple associated with A must have been generated before those associated with B or C. Four tuple orders will be allowed (abc, acb, bac, cab)[9]; the other two possible orders (cba, bca) will be rejected and the particular combinations of tuples exhibiting a disallowed order will not participate in the query[10]. The <when clause> and <where clause> work in concert to determine which tuples associated with the <tuple variable>s appearing in the query will be used to derive a result tuple.

3.5. The Temporal Delimiter Component

Time that takes survey of all the world
Must have a stop.

--*Shakespeare*, in Henry IV, Part 1

The temporal delimiter component specifies the value of the implicit time domain, just as the domain specification component identifies the values of the explicit domains. Two clauses, a <start clause> and a <stop clause>, are used when the result is a period relation; the <at clause> is used when the result is an event relation. The syntax is shown below:

---

[9] Here, the convention being used is that a represents a tuple associated with the <tuple variable> A, and similarly with b and c.

[10] Note that tuples must be supplied by all <tuple variable>s; hence, orders such as ab are not considered.

```
<temporal delimiter>    ::= <period delimiter> | <at clause>
<period delimiter>      ::= <start clause> <stop clause>
<start clause>          ::= ε | start <event expression>
<stop clause>           ::= ε | stop <event expression>
<at clause>             ::= at <event expression>
```

Using the relation defined earlier,

```
RunningOn (Processor, Process)
```

the queries

```
range of R is RunningOn
retrieve StartRunning (R.all)
at R.start
retrieve StopRunning (R.all)
at R.stop
```

specify the events which temporally delimit the RUNNINGON relation. The query

```
range of C is Clock(100)
retrieve SampledRunning (R.all)
at C
```

specifies an event relation with tuples at every 100 milliseconds designating which processes were running on which processors at that time. The tuples in the RUNNINGON relation valid at the time the clock ticks are placed in the SAMPLEDRUNNING relation. The monitor will implement this derived relation by sampling each processor every 100 milliseconds to determine which process is currently running.

3.6. Aggregate Operators

Quel uses the aggregate operators **count**, **sum**, **avg**, **min**, **max**, and **any** (the value is 1 if any tuples satisfy the qualification) to aggregate a computed expression over a set of tuples. The argument of such an operator can be either a single <tuple variable> or any expression involving constants, arithmetic operators, or domains of a single relation. The following query determines the total payroll (using the EMPLOYEE relation introduced at the beginning of this chapter):

```
range of E is Employee
retrieve (PayRoll = Sum(E.Salary))
```

The argument of the aggregate operator can be qualified; this query determines how many employees work in the toy department:

```
retrieve (Number=Count(E.Name where E.Dept = "Toy"))
```

Both queries are examples of *simple aggregates,* which evaluate to a single scalar value. *Aggregate functions,* on the other hand, partition the set of qualifying tuples into groups, each of which is assigned a value for the expression. The query

```
retrieve (E.Name, MS = Min(E.Salary by E.Dept))
```

returns a list of employees, each with the minimum salary of his or her department. Operationally, **avg** partitions the tuples into groups by department, then assigns a value (the average salary) to the tuples in the group. Each tuple receives the same value.

Aggregate operators are more complicated in TQuel, due to the time-varying behavior of relations. Aggregate operators on event relations are *cumulative,* in that they take all previously valid tuples into account in their computation. For instance, the **Count** operator on an event relation would count the number of events which had occurred. Figure 3-1a depicts a series of events. Figure 3-1b show the periods and their values for the query

```
retrieve (Value = Count(E))
```

As another example, the following query implements a simple clock:

```
range of C is Clock(1000)
retrieve (ClockTime = Count(C))
```

Since the clock "ticks" every second (1000 msec), there would be an event generated every second. The ClockTime domain would record the number of seconds that had passed before the current "tick".

There are two versions of aggregates on period relations, the cumulative and *instantaneous* versions. Since the instantaneous version is the default for aggregates applied to periods; the **CountC** operator is used to indicate the cumulative version, which works exactly as it does on event relations. The result of the (instantaneous) **Count** operator will in general go up and down as the periods come and go, while the value of the **CountC** operator must monotonically increase over time. In Figure 3-1c, a collection of periods is shown. Figure 3-1d shows the result of the **Count** operator, whereas Figure 3-1e shows the result of the **CountC** operator. Note that the derived periods of the **CountC** operator only involve the leading edge of the underlying periods, while those of the **Count** operator involve both edges.

The **avgc** operator is slightly different, since it takes the length of time the tuple was valid into account when computing the average. The value of the argument of the **avgc** operator is weighted by the duration of the tuple, and intervening periods (when no tuple is valid) are treated as tuples with a value of 0 for the argument.

Note that the presence of an aggregate operator in a **retrieve** statement automatically implies that the resulting relation will be a period relation. The <at clause> may be used to specify that an event relation is to be derived. The conversion from single event relations to period relations is handled by the **ExtendC** aggregate operator, which extends an event to a period stretching to the next event. It is cumulative since the derived period depends on the preceding event.

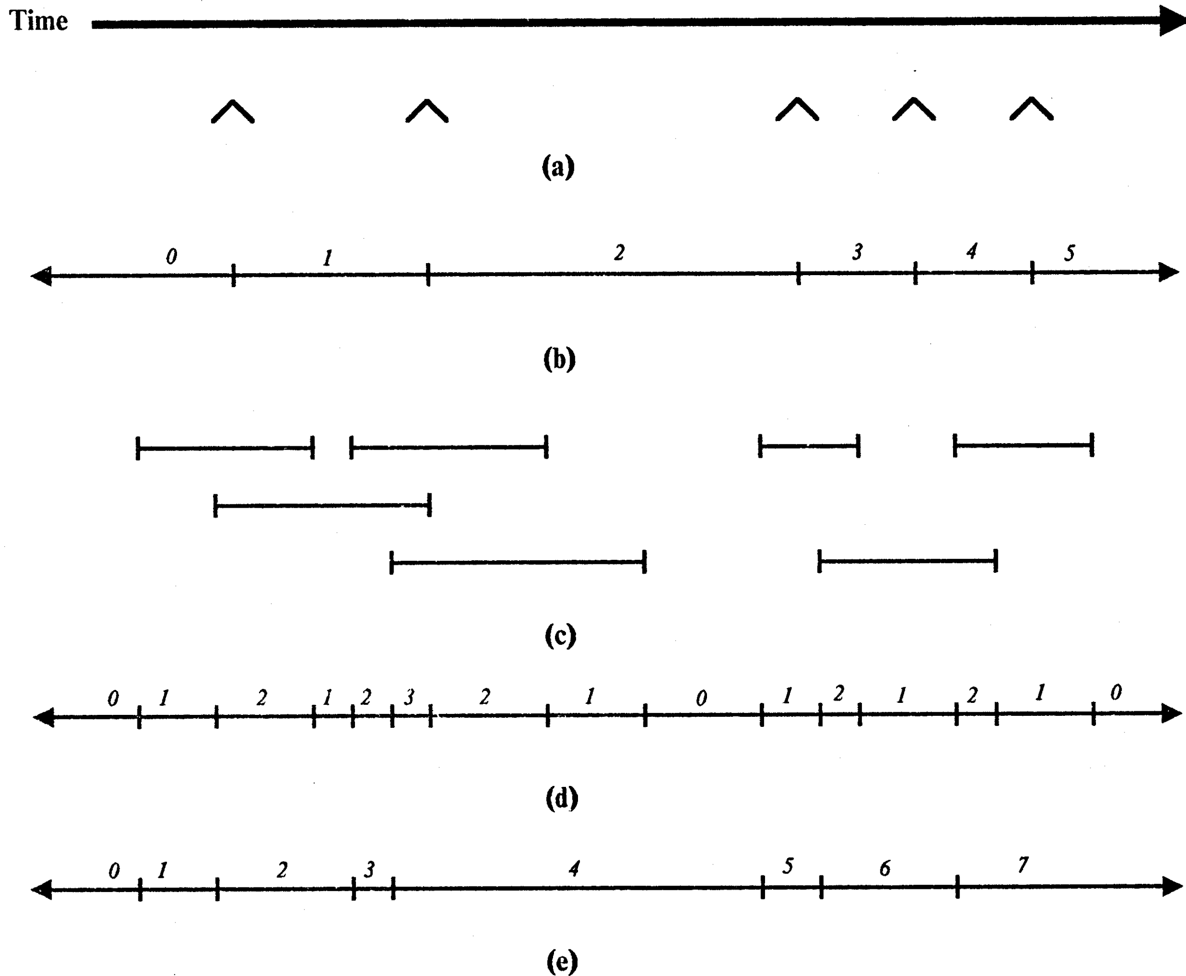

Figure 3-1: Instantaneous versus Cumulative Count

3.7. An Example

To illustrate the actions of the monitor, we will examine how a particular program running on Cm* is monitored. The program solves Laplace's partial differential equation with given boundary conditions (Dirichlet's problem) by the method of finite differences. The equation

$$\frac{\partial^2 u(x,y)}{\partial x^2} + \frac{\partial^2 u(x,y)}{\partial y^2} = 0$$

is solved for points on an m by n rectangular grid, where only the values at the outer edges of

the grid are given. The solution is found iteratively. On each iteration, the new value of each element is set to the arithmetic average of the values of its four adjacent neighbors.

Several processes and several processors work on the grid simultaneously. The grid is partitioned into regions, with one process responsible for each region. The configuration is shown in Figure 3-2. Note that the solvers require access to adjacent regions to derive new values for points on the boundary of their region.

There are many possible ways to synchronize the processes, the most efficient being the purely asynchronous method. The processes are only synchronized at the beginning of the computation. This means that, due to differences in the scheduling and in the data that each process is working on, some processes may perform many more iterations than others.

The proposed experiment will investigate the relative synchrony of two of the processes operating on adjacent regions. If one of the processes (call it P_1) gets behind the other process (P_2), then the second process will be using older values for the points on the boundary, possibly slowing the convergence for the entire grid. This experiment will focus on those periods of time when P_1 gets significantly behind P_2 (i.e., more than one iteration).

One sensor is needed, a traced event sensor which generates an event record each time the solver process begins a new iteration. Since this sensor, called Iteration, is traced, the events are automatically converted into a primitive period relation (also called ITERATION) by the monitor. Thus, the primitive period Iteration is defined, with two domains: Process, the name of the process generating the event record, and IterNum, an integer designating the iteration which has just begun. Derived relations can now be specified using the Iteration relation.

A period begins whenever the IterNum domain changes, with the tuples partitioned into groups according to the Process domain. The query

```
range of A is Iteration
range of B is Iteration
retrieve AOverB (Diff = B.IterNum - A.IterNum)
where A.Process = P1 and B.Process = P2
        and A.IterNum > B.IterNum + 1
```

finds the periods of time in which P_2 is behind P_1 by at least 2 iterations. The start and stop clause>s default to the most conservative situation (see section 4.5.2), that is, the result tuple will be valid only as long as both underlying tuples were valid.

The periods in the AOVERB relation represent the times where process P_2 was significantly behind process P_1. To determine the percentage of time this was the case, use the query

```
range of AB is AOverB
retrieve Over (Percent=AvgC(AB) * 100)
```

The aggregate operator used here is the cumulative average operator. Since a single <tuple

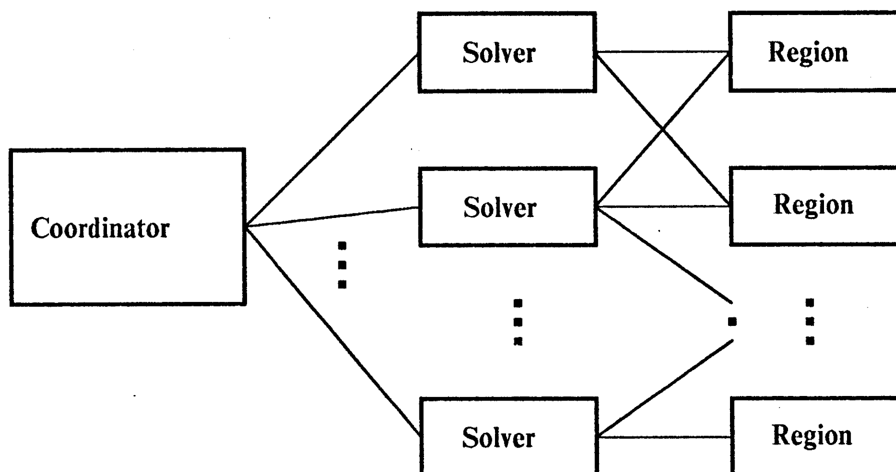

Figure 3-2: Configuration of the PDE task force

variable> appears as an argument to the **avgc** operator, the result will range from 0 (no tuples ever occurred) to 1 (tuples are always present).

And finally, to determine the times P_2 caught up with P_1,

```
retrieve Catch
where A.Process = P1 and B.Process = P2
        and A.IterNum = B.IterNum
when A.Start ; B.Start
at B.start
```

Since no target domains were specified, the CATCH relation will only contain the implicit time domain. The <when clause> says that P_2 started a new iteration during an iteration of P_1. Since the IterNum is always increasing, P_2 (B) can catch up with P_1 only in this manner. The alternative,

```
when B.start ; A.start
```

specifies that the iteration numbers were equal after P_1 has started a new iteration, implying that P_2 was already ahead. The <at clause> indicates the exact time that P_2 does catch up. Of course, P_1 will probably start its next iteration shortly, leaving P_2 behind once again.

Finally, to view the results of these derivations,

```
display Over, Catch
```

3.8. Summary

Date lists eight criteria to be applied in the evaluation of conceptual views of data which are also helpful in evaluating query languages [Date 76]. It is useful to examine TQuel in relation to these criteria, and, more specifically, in relation to the language from which it was derived, Quel.

1. *The number of basic constructs should be small.*

   The Quel **retrieve** statement consists of a domain specification component (the <target list>) and a tuple selection component (the <where clause>). The TQuel **retrieve** statement consists of precisely the same components, so at this level, the number of basic components has remained small.

   At the syntactic level, TQuel augments the <target list> and <where clause> with (a) the <at clause>, <start clause>, and <stop clause> (temporal analogues of the <target list>) and (b) the <when clause> (the temporal analogue fo the <where clause>). Hence, the syntax is more complex, but not overly so.

2. *Distinct concepts should be cleanly separated.*

   The most important example is the distinction between the temporal delimiter component and the temporal selection component. Although the expressions found in these two components may be quite similar, the very dissimilar semantics is made clear through the use of separate reserved words (**start**, **stop**, and **at** versus **when**).

3. *Symmetry should be preserved.*

   Symmetry should arise between the explicit domains and the implicit temporal domains, as well as between the start and stop domains. In the former case, both sets of domains are associated with a domain specification component and a tuple selection component. In the latter case, TQuel associates quite similar syntax and semantics with the start and stop domains.

4. *Redundancy should be controlled.*

   This objective is concerned more with the information being stored in the database than with the query language.

5. *The number of operator types should be small.*

   TQuel adds only a few temporal operators (three binary and three unary) and one aggregate operator. Much of the language design involved incorporating time into the semantics of the existing operators.

6. *Very high-level operators should be available.*

   The sequence (;), alternation (|), and concurrency (,) operators are high-level in that they indicate *which* properties the resulting tuples should have, rather than specifying *how* the properties are to be obtained.

7. *Behavior must be totally predictable and should accord with the user's intuitive expectations.*

   This objective is dealt with in detail in the next chapter, especially with regard to aggregates, defaults, and semantics when the time domain is fixed.

8. *The language should be founded conceptually on a solid base of theory.*

   The presence of a complete formal semantics for the TQuel **retrieve** statement, as presented in the next chapter, satisfies this objective.

In summary, the Quel language has been augmented syntactically and semantically to incorporate time. The syntactic changes included four new keywords, **when**, **start**, **stop**, **at**; several new functions; and two new types of expressions, <temporal expression>s and <event expression>s. Semantic changes included a new domain type, temporal; additional selection and specification components; and additional semantics for aggregate operators. Examples indicate that TQuel allows complex queries to be specified in a straightforward manner with little irrelevant detail. Hence, TQuel is an existence proof that

> **Result:** Traditional query languages can be augmented syntactically and semantically to include time.

This introduces a new issue, to be addressed in the next chapter:

> **Problem:** How can the semantics of TQuel be formalized?

Chapter 4
Semantics of the TQuel Retrieve Statement

The story is told of the Russian poet Samuel Marshak that when he was first in London, and did not know English very well, he went up to a man in the street and asked, 'Please, what is time?' The man looked very surprised and replied, 'But that's a philosophical question. Why ask me?'

--*G. J. Whitrow*, in The Nature of Time

It is impossible to meditate on time ... without an overwhelming emotion of the limitations of human intelligence.

--*A. N. Whitehead*

4.1. Tuple Calculus

The semantics of the TQuel retrieve statement is an extension of that of the standard Quel retrieve statement. The semantics of both will be given as tuple calculus expressions, which are of the form

$$\{ t^{(i)} \mid \psi(t) \}$$

where the variable t denotes a tuple of some fixed length *i*, and $\psi(t)$ is a first order propositional calculus expression containing only one free tuple variable t. $\psi(t)$ defines the tuples contained in the relation specified by the retrieve statement. The atoms of ψ are of three types:

- R(s), where R is a relation name and s is a tuple variable, asserting that s is a tuple in relation R;

- $s[i]\ \theta\ u[j]$, where s and u are tuple variables and θ is an arithmetic comparison operator, asserting that the i^{th}component of s stands in relation θ to the j^{th}component of u; and

- s[i] θ a and a θ s[i], where a is a constant, having a similar meaning.

For example, the intersection of R and S (both of arity i) is expressed by the calculus expression

$$\{ t^{(i)} \mid R(t) \wedge S(t) \}$$

A more detailed presentation of tuple calculus can be found in [Ullman 82].

The Quel statement

```
range of t1 is R1
...
range of tk is Rk
retrieve ( ti1 . D1 , ..., tir . Dr)
where ψ
```

is equivalent to the tuple calculus statement

$$\{ u^{(r)} \mid (\exists t_1) \cdots (\exists t_k) (R_1 (t_1) \wedge \cdots \wedge R_k (t_r)$$
$$\wedge\, u[1] = t_{i_1} [j_1] \wedge \cdots \wedge u[r] = t_{i_r} [j_r]$$
$$\wedge\, \psi') \}$$

which states that t_i is in R_i, that the result tuple u is composed of r particular components of the t_i's, that D_m is the j_mth attribute of the relation R_{i_m}, and that the condition ψ (modified trivially for domain names and some Quel syntax conventions) holds for u. In the first example given in the previous chapter (using the EMPLOYEE (Name, Dept, Salary, Manager, Age) relation),

```
range of E is Employee
retrieve into T (Comp = E.Salary / (E.Age - 18))
where E.Dept = "Toy"
```

the corresponding tuple calculus statement is

$$\{ u^{(1)} \mid (\exists E) (\text{Employee}(E)$$
$$\wedge\, u[1] = E[3] / (E[5] - 18)$$
$$\wedge\, E[2] = \text{"Toy"}) \}$$

The result is a set of single domain tuples, each with the property that the domain is computed from the third and fifth domains of a tuple from the EMPLOYEE relation which has a value for the second domain equal to "Toy". In the remainder of this chapter, domain names, rather than domain indices, will be used. Hence, this statement can be rewritten

$$\{ u^{(1)} \mid (\exists E) (\text{Employee}(E)$$
$$\wedge\, u[\text{Comp}] = E[\text{Salary}] / (E[\text{Age}] - 18)$$

$\wedge$ E[Dept] = "Toy")}

To review, the primary additions incorporated into TQuel were the selection (when) and specification (start, stop, and at) components. The temporal expressions in the when clause serve to select tuples to participate in the rest of the query. The temporal expression specifies an ordering on the tuples. Conceptually, each possible set of tuples, one for each tuple variable, is applied to the temporal expression. If the tuples satisfy the specified ordering, they will be used for further processing in the query. The specification component uses event expressions yielding events instead of a Boolean. In this framework, every clause in the event expression takes one or more events or periods and yields a value which is either an event or a period, with the complete event expression yielding an event.

4.2. Path Expressions in TQuel

The syntax of both temporal and event expressions is drawn from path expressions. Path expressions were originally proposed as a high-level synchronization construct specifying the allowable sequences of operations on an object of an associated abstract data type. The following is a list of path expression constructs, with α and β representing path expressions and ω denoting an operation on an object:

empty path	ε	
elementary operation	ω	
parallel	α , β	α occurs in parallel with β
sequence	$\alpha ; \beta$	α is followed in time by β
selection	$\alpha \mid \beta$	either α or β occurs
repetition	$\alpha +$	one or more consecutive execution sequences of α
	α^*	$(= \varepsilon \mid \alpha +)$ zero or more consecutive execution sequences of α
concurrency	$\omega \#$	one or more concurrent execution sequences of ω

When path expressions are used in TQuel, the operations are replaced by tuple variables. The path expression then specifies a (Boolean or event) value derived from the tuples as-

sociated with the tuple variables appearing in the expression. If only the selection, sequence, and parallel operators appear in the expression, then one tuple is associated with each tuple variable. This has the same semantics as the Quel retrieve statement, since each resulting tuple is derived from one tuple associated with each tuple variable appearing in the query. If the repetition or concurrency unary operators are used, then each tuple variable appearing in the expressions involved with these operators can be associated with multiple tuples. For instance, the path expression

A*

specifies a sequence of non-overlapping tuples, all associated with the tuple variable A and all participating in the derivation of a single result tuple. Allowing multiple tuples per tuple variable complicates the processing of the query. More importantly, the semantics for the Quel retrieve statement given earlier is predicated on one tuple per tuple variable (per output tuple). Therefore, allowing multiple input tuples per tuple variable greatly alters the semantics of the expression. The assumption will be that there is one tuple per tuple variable, and thus that no temporal or event expression contains repetition ('+' and '*') or concurrency ('#') operators.

The semantics for the two uses of path expressions, returning a Boolean and returning an event, will be dealt with separately. Returning an event is closer semantically to expression evaluation, and will thus be considered first.

4.2.1. The Start, Stop, and At Clauses

As discussed previously, the temporal delimiter component specifies the time during which the derived tuple is valid. For derived periods, the start and stop clauses are used; for derived events, the at clause is used. In all three clauses, an event expression is used to specify an event. It is important to note that the event returned by the event expression will in fact be one of the events originally involved in that expression. Hence, the event expression is not actually *deriving* a new event from the given events; rather, it is *selecting* one of the given events to return as a value. Of course, the selection criteria can be, and indeed usually is, a function of the relative temporal ordering of the original events.

When full path expressions are allowed in event expressions, they can be ambiguous when returning an event. The problem is the selection operator. There are two possibilities for interpreting the following temporal expression as returning an event:

$\alpha \mid \beta$

Either the system must ensure that α and β are disjoint, so that only one will return a value, or ensure that both yield the same value. Hence, the temporal expression

(a ; (b | c)) . stop

must not be allowed, for the selection terms are not disjoint, nor do they yield the same value. If the temporal ordering of the tuple variables was bac, then this expression would return the event associated with c (b is ignored in this case). However, if instead the temporal ordering was abc, the event associated with either b or c could be returned.

Determining whether the selection terms are disjoint or return the same value is difficult. The solution is to not allow the selection operator in event expressions.

One aspect remains to be specified; what is the value of "(α , β)"? There are two reasonable interpretations; the overlap (o) interpretation, where the result is valid only when *both* underlying tuples are valid, and the coverage (c) interpretation, where the result is valid if *either* of the underlying tuples is valid. The difference between the two interpretations is illustrated in Figure 4-1. Although the overlap interpretation seems more natural, and was used in the previous chapter, the distinction from a semantic viewpoint is minimal, so semantics for both interpretations will be presented.

The syntax of event expressions defines a parse tree containing the following node types: <tuple variable>, <.start>, <.stop>, <parallel>, and <sequence>. The <tuple variable> nodes are the leaves of the parse tree; the <.start> and <.stop> nodes have one descendant, and the <sequence> and <parallel> nodes have two descendants. This tree can be executed directly in a bottom-up fashion: the <tuple variable> nodes will yield either an event (i.e., a timestamp) or a period (a pair of timestamps); the <.start> and <.stop> nodes will take a period and yield an event (the first or second of the timestamps, respectively); the <sequence> node will accept two periods or events and yield a period. The action of the parallel node will depend on whether the overlap or coverage interpretation is used; in either case, its action is well-defined. The result of the top node of the tree will be an event from one of the tuple variables in the event expression.

The semantics is now straightforward to specify. Each event expression can be transformed into an expression on the terminals using the parse tree; the expression will contain the following functions (E ranges over timestamps):

Start: $E^2 \rightarrow E$

Stop: $E^2 \rightarrow E$

Sequential$_1$: $E \times E \rightarrow E^2$

Sequential$_2$: $E \times E^2 \rightarrow E^2$

Sequential$_3$: $E^2 \times E \rightarrow E^2$

Parallel(c)$_1$: $E \times E \rightarrow E^2$

Parallel(c)$_2$: $E^2 \times E \rightarrow E^2$

Parallel(c)$_3$: $E \times E^2 \rightarrow E^2$

Parallel(c)$_4$: $E^2 \times E^2 \rightarrow E^2$

Parallel(o)$_1$: $E \times E^2 \rightarrow E$

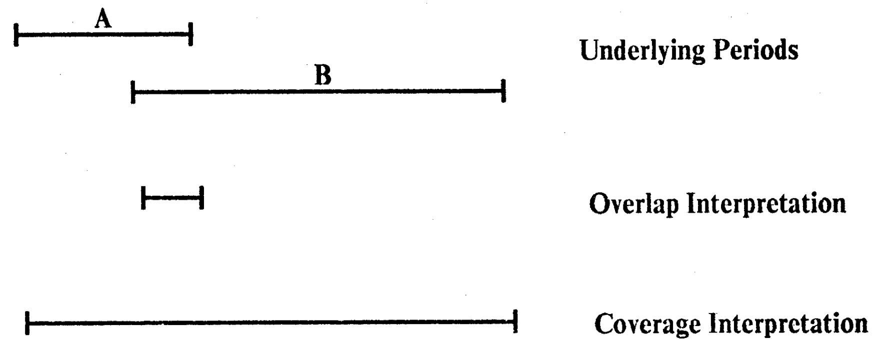

Figure 4-1: The Overlap versus Coverage Interpretations for Combining Periods

$\text{Sequential}_4 : E^2 \times E^2 \rightarrow E^2$

$\text{Parallel(o)}_2 : E^2 \times E \rightarrow E^2$

$\text{Event: Tuple Variable} \rightarrow E$

$\text{Parallel(o)}_3 : E^2 \times E^2 \rightarrow E^2$

$\text{Period: Tuple Variable} \rightarrow E^2$

There are four separate sequential functions, to accommodate all possible combinations of events and periods: (event ; event), (event ; period), (period ; event) and (period ; period). All yield periods. The parallel operator under the overlap interpretation has no semantics for (event , event). Event expressions under this interpretation will consider such cases to be errors.

Let Φ_τ be the function resulting from the transformations on τ. The use of Φ_τ will be examined after the semantics of the when clause has been discussed.

4.2.2. The When Clause

Several researchers have proposed a formal semantics for particular variations on path expressions, involving denotational and axiomatic definitions [Berzins&Kapur 77], or transformations into Petri nets [Lauer&Campbell 75] or parallel programs [Andler 79]. The approach taken here transforms the temporal expressions directly into a set of execution histories on the tuple variables involved in the expression. Each execution history specifies a valid ordering of the tuples referenced by the expression. For example, the temporal expression, where A, B, and C are tuple variables denoting event relations,

(A ; B) , C

will be translated into the set

{ ABC, ACB, CAB }

An assignment of tuples to the tuple variables, when ordered by time, must correspond to one of the execution histories in the set. Hence, by providing the transformations and proving that they yield a set of execution histories when applied to a temporal expression, we will have defined the semantics of such expressions.

The syntax of the temporal expressions is as follows:

```
<t-exp>    ::= <element>
             | <t-exp> . time
             | <t-exp> . start
             | <t-exp> . stop
             | <t-exp> ; <t-exp>
             | <t-exp> "|" <t-exp>
             | <t-exp> , <t-exp>
             | ( <t-exp> )
```

Before the transformations are made to the temporal expressions, all tuple variables associated with period relations are replaced with (P.start ; P.stop), where P is the relevant tuple variable. All instances of P . start, P . stop, E, and E . time, where E is any tuple variable associated with an event relation, will be referred to as *terminals.* Each terminal specifies an event in the temporal expression.

The transformation system defining the semantics of temporal expressions is comprised of seventeen productions (where a and b are terminals, and α, β, and γ are arbitrary temporal expressions):

(1) a . start $\Rightarrow$ a

(2) $(\alpha \mid \beta)$. start $\Rightarrow$ $(\alpha$. start $\mid \beta$. start)

(3) $(\alpha ; \beta)$. start $\Rightarrow$ α . start

(4) (α , β) . start $\Rightarrow$ $(\alpha$. start ; β . start) | (β . start; α . start)

(5) a . stop $\Rightarrow$ a

(6) $(\alpha \mid \beta)$. stop $\Rightarrow (\alpha \mid \beta)$

(7) $(\alpha ; \beta)$. stop $\Rightarrow \beta$. stop

(8) (α , β) . stop $\Rightarrow (\alpha$. stop $\mid \beta$. stop$)$

(9) (α) $\Rightarrow \alpha$

(10) a , $(\beta ; \gamma)$ $\Rightarrow$ a

(11) $\alpha , (\beta \mid \gamma)$ $\Rightarrow (\alpha , \beta) \mid (\alpha , \gamma)$

(12) $(\alpha \mid \beta) , \gamma$ $\Rightarrow (\alpha , \gamma) \mid (\beta , \gamma)$

(13) $(a ; \alpha) , b$ $\Rightarrow (a ; b ; \alpha) \mid (a ; (\alpha, b))$

(14) $(a_1 ; \cdots ; a_j) , (b_1 ; \cdots ; b_k)$ $\Rightarrow (a_1 ; b_1 \mid b_1 ; a_1) ; (a_j \mid b_k)$

(15) $\alpha ; (\beta \mid \gamma)$ $\Rightarrow (\alpha ; \beta) \mid (\alpha ; \gamma)$

(16) $(\beta \mid \gamma) ; \alpha$ $\Rightarrow (\beta ; \alpha) \mid (\gamma ; \alpha))$

Since these productions define the meaning of temporal expressions, it is important that the reader is convinced that each production matches his or her intuitive understanding of the various constructs. Productions (1), (2), (5), (15), and (16) are straightforward. Production (9) is used to remove extraneous parentheses; α must not be a binary expression to apply this production.

Production (3) can best be understood when examined as part of a larger temporal expression:

$(\alpha ; \beta)$. start ; δ

expresses two possible constraints on the ordering of α, β, and δ: (a) β must follow α in time, and (b) δ must follow the beginning of the period begun when α started and ended when β finished. Since . start was explicitly specified, only constraint (b) is taken as the semantics for this expression. To express constraint (a), the user must add the clause

$\wedge (\alpha ; \beta)$

to the expression. Constraint (b) is indicated by replacing the subexpression $(\alpha ; \beta)$. start with α . start. Therefore, the production would map the example into

$(\alpha\ .\ \text{start}\ ;\ \delta\ )$

Similar comments apply to production (7). With regard to production (6), consider

$(\alpha\ |\ \beta)\ .\ \text{stop}\ ;\ \delta$

This states that the end of either α or β must be followed by δ. This statement is equivalent to saying that either α or β must be followed by δ, or

$(\alpha\ |\ \beta)\ ;\ \delta$

Productions (4) and (8) are initially confusing: although the left hand sides are similar, the right hand sides look quite different. As with previous productions, their effect can be understood in terms of their containing expression. For production (4), examine this expression

$(\alpha\ ,\ \beta)\ .\ \text{start}\ ;\ \delta$

α and β occur in parallel; δ must follow the beginning of this parallel activity. In the overlap interpretation, $(\alpha\ ,\ \beta)$ is valid at the point when *both* α and β become valid. Since α . start and β . start are both events (independent of the internal structure of α and β), they can be ordered:

$$(\alpha\ ,\ \beta)\ .\ \text{start} \equiv ((\alpha\ .\ \text{start}\ ;\ \beta\ .\ \text{start})\ |\ (\beta\ .\ \text{start};\ \alpha\ .\ \text{start}))\ .\ \text{stop}$$

$$\Rightarrow (\alpha\ .\ \text{start}\ ;\ \beta\ .\ \text{start})\ |\ (\beta\ .\ \text{start}\ ;\ \alpha\ .\ \text{start}) \qquad \text{(by (6))}$$

Similarly, for production (8), examine

$(\alpha\ ,\ \beta)\ .\ \text{stop}\ ;\ \delta$

In the overlap interpretation, δ may start when *either* α or β have stopped. Again, since α . stop and β . stop are events,

$$(\alpha\ ,\ \beta)\ .\ \text{stop} \equiv ((\alpha\ .\ \text{stop}\ ;\ \beta\ .\ \text{stop})\ |\ (\beta\ .\ \text{stop}\ ;\ \alpha\ .\ \text{stop}))\ .\ \text{start}$$

$$\Rightarrow (\alpha\ .\ \text{stop}\ |\ \beta\ .\ \text{stop}) \qquad \text{(by (2) and (3))}$$

The next six productions define the semantics of the parallel operator interacting with the sequence and selection operators. Except for (11) and (12), which allow distribution of the parallel operator over selection, the productions all involve at least one terminal. The reasoning behind these productions can be seen by considering a production *not* on this list:

$$\alpha\ ,\ (\beta\ ;\ \gamma) \rightarrow ((\alpha\ ,\ \beta)\ ;\ \gamma)\ |\ (\beta\ ;\ (\alpha\ ,\ \gamma))\ |\ ?$$

The problem is that α may occur in parallel with β, in parallel with γ, or in parallel with both (remember that α can have an arbitrary internal structure). (10) substitutes the terminal a, which is guaranteed not to have any internal structure, for α. In the overlap interpretation the result of an event occurring in parallel with a period is simply that event. Of course, a side condition is that a must occur after β . start and before γ . stop. As with (3) and (7), this condition must be stated explicitly by the user.

(13) and (14) also use this technique. In (13), b can either be between a and the start of α,or during α.

Production (14) is a generalization of (13), and is thus more complex. In the overlap interpretation, the result starts as soon as *both* sequences start, and ends when *either* sequence end. Since a_1 and b_1 (and a_j and b_k) are events, they temporally delimit the respective sequences. They can also be ordered. Hence

$$(a_1 ; \cdots ; a_j) , (b_1 ; \cdots ; b_k) \equiv (a_1 ; b_1 \mid b_1 ; a_1) . \text{stop} ; (a_j ; b_k \mid b_k ; a_j) . \text{start}$$
$$\Rightarrow (a_1 ; b_1 \mid b_1 ; a_1) ; (a_j \mid b_k) \qquad \text{(by (6), (2), (3))}$$

The usefulness of these productions is evident in the following theorem:

Conversion Theorem:

The productions (1) through (16) transform an arbitrary temporal expression into a relational expression (involving only the sequence and selection operators) in standard form (a selection of sequences):

$$(a_1 ; a_2 ; \cdots ; a_i) \mid (b_1 ; \cdots ; b_j) \mid \cdots \mid (z_1 ; \cdots ; z_k)$$

The proof is somewhat involved, and is given in appendix B. The standard form provides the set of execution histories defined by the original temporal expression.

4.3. Formal Semantics

It is now possible to specify a formal semantics for the TQuel retrieve statement. Let τ' be the temporal expression τ with the tuple variables t_i which correspond to periods replaced with (t_i . start ; t_i . stop), the t_1 . start terms replaced with δ_i, the t_i . stop terms replaced with δ_{i+k}, and the t_i . time terms (and t_i terms, for those tuple variables corresponding to events) replaced with δ_i. The δ_i serve as terminals in the rest of the analysis. This somewhat unusual numbering scheme is necessary in order to make t_i . start and t_i . stop unique terminals. Define

$\tau(\delta_i) \triangleq \tau_i$ [starttime] if $i \leq k$, and τ_i [stoptime] otherwise

The starttime and stoptime domains are the implicit domains associated with all tuples. Let

Order (τ') = $\delta_{m_1} \cdots \delta_{m_n}$, where n is the number of unique terminals in τ',

τ' contains δ_{m_i}, and $\tau(\delta_{m_i}) \leq \tau(\delta_{m_{i+1}})$, for $1 \leq i \leq n - 1$

Order defines a sequence of terminals ordered by the time values of the tuples associated with those terminals. This definition of temporal order assumes that *metric time* is being used, where time is modeled as the real number line, and the "before" relation is isomorphic to "<" for reals (i.e., the time of Newtonian physics). In the context of monitoring distributed sys-

tems, the measured time must be global and satisfy the before relation. Finally, let $W(\tau)$ be the set of sequences generated from τ by the productions as shown by the theorem.

With all of this mechanism, the semantics of the TQuel retrieve statement

```
range of t1 is R1
...
range of tk is Rk
retrieve (ti1 . D1 ... tir . Dr )
where ψ
when τ
start ν
stop χ
```

is quite straightforward:

$$\{ u^{(r+2)} \mid (\exists t_1) \ldots (\exists t_k) (R_1 (t_1) \wedge \ldots \wedge R_k (t_k)$$
$$\wedge\, u[1] = t_{i_1} [j_1] \wedge \ldots \wedge u[r] = t_{i_r} [j_r]$$
$$\wedge\, \psi'$$
$$\wedge\, \mathrm{Order}(\tau') \in W(\tau')$$
$$\wedge\, u[\mathrm{starttime}] = \Phi_\nu (t_{k_1}, \ldots, t_{k_p})$$
$$\wedge\, u[\mathrm{stoptime}] = \Phi_\chi (t_{m_1}, \ldots, t_{m_q})$$
$$) \}$$

Note that p is defined to be the number of unique tuple variables in ν, and q the number of unique tuple variables in χ. Φ_ν was defined in section 4.2.1 to be the function derived from the event expression ν on the tuple variable in the expression. The superscript $(r + 2)$ indicates that the tuple u has r explicit domains and 2 implicit domains, the starttime and stoptime domains (events will have only one implicit domain).

The when clause specifies that the tuples, when ordered temporally, correspond to one of the execution histories in the set of execution histories defined by the temporal expression. The start and stop clauses specify the values of the starttime and stoptime domains through the functions defined by the event expressions appearing in those clauses.

There are two aspects remaining to be covered: aggregates and indeterminacy. Although the tuple calculus semantics for Quel retrieve statements without aggregate operators may be found in [Ullman 82], no such semantics is given for the more general case. Indeterminacy is totally absent from Ingres and Quel; the information in the database is assumed to be consistent and complete for the aspects of the real world at a particular time being modeled by the database. Such an approach is infeasible when using temporal databases in monitoring. First, a semantics will be developed for aggregates, and then all of the semantics will be extended to include indeterminacy.

4.4. Aggregate Operators

The semantics of standard Quel aggregate operators (c.f., section 3.6) is best handled by defining the result of the aggregate, and then using this result in the tuple calculus expression for the entire statement (the implementation performs an analogous extraction of the aggregate expressions). Simple aggregates result in a scalar value; using an example from the previous chapter:

retrieve (Number = Count(E.Name where E.Dept = "Toy"))

$\Rightarrow$ Num = cardinality of

$$\{ n^{(1)} \mid (\exists E)(\text{Employee}(E) \wedge E[\text{Dept}] = \text{"Toy"} \wedge n[1] = E[\text{Name}]) \}$$

$$\{ u^{(1)} \mid (\exists E)\ (\text{Employee}(E) \wedge u[\text{Number}] = \text{Num}\}$$

Since the tuple variable did not appear outside of an aggregate expression, it can be eliminated in the second tuple calculus expression:

$$\{ u^1 \mid u[\text{Number}] = \text{Num} \} \Rightarrow \{ \text{Num} \}$$

Aggregate functions require intermediate relations:

retrieve (E.Name, MS = Min(E.Salary by E.Dept))

$$\Rightarrow \text{MS} = \{ m^2 \mid (\forall E)\ ((\text{Employee}(E) \wedge E[\text{Dept}] = m[1]) \supset m[2] \leq E[\text{Salary}])$$
$$\wedge (\exists F)\ (\text{Employee}(F) \wedge F[\text{Dept}] = m[1] \wedge m[2] = F[\text{Salary}]) \}$$

This statement defines a relation MS, with two domains, a department and a minimum salary. The first clause states that all employees of that department make at least the minimum salary of the department; the second clause says that at least one employee does indeed make the minimum salary. Both clauses are necessary to correctly define the minimum salary. The calculus statement defining the result of the retrieve statement uses this temporary relation:

$$\{ u^2 \mid (\exists E)(\exists m)\ (\text{Employee}(E) \wedge \text{MS}(m) \wedge m[1] = E[\text{Dept}]$$
$$\wedge u[\text{Name}] = E[\text{Name}] \wedge u[\text{MS}] = m[2] \}$$

The correct minimum salary is selected from the MS relation using the department domain. It is evident that the semantics for an arbitrary Quel aggregate would be similar to the above expressions.

As with the other constructs, the semantics of the TQuel aggregate operators will be obtained by extending the tuple calculus statements just presented. The central issue is how to incorporate time into the predicates of the calculus statement. Before this issue can be addressed, however, the informal semantics must be well-understood.

4.4.1. Informal Semantics

Extending Quel aggregates to include the passage of time is surprisingly complex. The most important goal is to ensure that the semantics are as natural as possible. A second goal is to ensure that, when time is stopped, say, by considering an aggregate at a single instant of time, the semantics are equivalent to the standard Quel semantics (i.e., they reduce to the Quel tuple calculus equivalents).

In Quel, the aggregate operator partitions the tuples into groups, and then applies the operator to each group. In TQuel, the composition of the groups changes over time (by adding and removing tuples), so the value of the function also changes over time. The value returned by an aggregate operator may be *instantaneous,* that is, derived at each instant in time from the values of tuples valid at that particular time, or *cumulative,* derived potentially from tuples valid at previous instances of time. The instantaneous version of **Count**, for instance, determines the number of tuples valid at each instant of time. The cumulative version of **Count** has as a value at a particular instant of time the number of tuples which are valid at that time, or were valid previously. The cumulative version increases monotonically, whereas the instantaneous version does not. Aggregate operations on event relations must be cumulative, since the probability that two or more events occur simultaneously becomes arbitrarily small as the timestamp granularity decreases. For period relations, two versions of each aggregate operator are provided. Note that for cumulative aggregate operators, there must be some designation of when time started. For instance, the cumulative count will be zero until the first period occurs. The initial tuple will have a value of 0 starting at the designated time and ending when the first tuple starts.

There is one other aspect concerning aggregate operators on temporal relations which should be addressed. The instantaneous **Count** operator can count periods easily. However, how does one cumulatively count a collection of periods? More specifically, if one period starts at t_1 and ends at t_2, and another period with exactly the same values for the domains starts at t_2 and ends at t_3, then should the count be 1 or 2?

This situation is similar to the issue of duplicate tuples in Ingres. Certain operators such as projection often generate duplicate tuples. These duplicates are usually tolerated, since eliminating them can be computationally expensive. However, aggregate operators such as **Count** will return different results depending on the presence of duplicate tuples. Ingres provides two versions of **count**, **sum**, and **avg**. **Count** returns the number of (possibly duplicate) occurrences and **CountU** returns the number of unique occurrences. Note that the other aggregate operators (**min**, **max**, **any**) do not have this problem. Hence, there is only one version of these operators.

In TQuel, there are two dimensions of duplication: identical tuples valid at the same instant of time, and identical tuples (in the explicit domains) valid at different instants (or periods) of

time. In the first dimension, relevant to instantaneous aggregates, duplicates are eliminated (e.g., only the analogue to the **CountU** operator is provided). In the second dimension, the temporal dimension, the issue is more complex.

There are at least three possible results for the situation of a period starting at the same time the previous period ended. The most straightforward is 2, since there are 2 tuples. This solution corresponds to the Ingres **Count** operator for the similar situation of duplicate tuples. The second possible answer is 1, since there was one contiguous period of time for which the relationship was valid. This approach would count the number of contiguous periods of time, with the periods separated by intervals when a tuple was *not* valid. This solution is analogous to the Ingres **CountU** operator. Implementation of this approach on a temporal database is difficult in the presence of indeterminacy. The third answer is neither, that the value of count in the integral of the number of periods over time. This solution is not strictly necessary for the **count** operator, since the same effect could be obtained using **sum(duration**(T)), where T is a <tuple variable>. Similarly, the integral **sumc** operator on T.Domain can be obtained using

```
T.Domain * sum(duration(T))
```

In chapter 3, the simplest option to implement, solution 1, was chosen.

There exists a potential **avgc** operator for every form of **sumc** and **countc**, since **avgc** is defined to be **sumc / countc**. In fact, even when only the integral characterization is used, there is some choice involved. There is one instantaneous version, and at least four versions of the integral **avgc** operator possible. The instantaneous version exhibits changes at both the start and stop events of the tuple. Versions 1 and 2 of the integral **avgc** operator exhibit changes in the aggregate whenever a new tuple begins, and ignore the duration of the tuple. They differ in whether a period starting at the same time as the previous period ends is considered one or two tuples. Version 3 weights the value by the duration of the tuple. However, the time periods when no tuple is valid are ignored, causing the resulting value of remain constant over those periods. Version 4 treats the intervening periods as tuples with a value of 0. Hence, the value of this version of **avgc** will, in the absence of tuples, asymptotically approach 0. Version 4 was chosen, since it seemed to be the most intuitive when applied to sample queries.

4.4.2. Formal Semantics

Now that a clear definition of the aggregate operator has been given, it is appropriate to be more formal. Recall that each tuple contains one or two implicit time domains, starttime and stoptime. These domains can be used to define a predicate indicating when a tuple was valid:

$$\text{valid}(u, t) \triangleq u[\text{starttime}] \leq t \leq u[\text{stoptime}]$$

Since time is represented using timestamps, this predicate involves comparing three real numbers (again, metric time is assumed). The instantaneous Count operator can now be defined as returning the number of tuples valid at any instant of time. Formalizing this statement in an example:

```
retrieve (Number = Count(E.Name where E.Dept = "Toy"))
```

$$\Rightarrow Num_t = \text{cardinality of}$$

$$\{ n^{(1+2)} \mid (\exists E)(Employee(E) \wedge valid(E,t) \wedge E[Dept] \leq \text{"Toy"}$$

$$\wedge\ n[1] = E[Name]) \}$$

$$\{ u^{(1+2)} \mid (\forall t)(valid(u, t) \supset u[Number] = Num_t \}$$

The first statement defines a scalar function Num_t, the number of employees in the toy department at time t. The second statement specifies that the Number domain of u contains the number of employees in the toy department during the time u is valid. Unfortunately, the period of time u is valid has been incompletely specified; in particular, the result relation should have only one tuple for every period of time when the count is constant, rather than several (perhaps overlapping) tuples containing the same count. The statement can be amended by defining the macro

$$Maximal\ (u, t, V) \triangleq (\exists t')(t' \neq t \wedge \neg valid(u, t') \wedge V_t = V_{t'})$$

$$\supset (\exists t'')((t < t'' < t') \vee (t > t'' > t')) \wedge V_t \neq V_{t''})$$

This macro ensures that u is valid for the entire period when V has the same value, by specifying that, given u is valid at time t and has a value V_t for the relevant domain, if u is not valid at another time t′, yet has the same value, then there must have existed an intermediate time when the value was different. The Maximal predicate can then be used as follows:

$$\{ u^{(1+2)} \mid (\forall t)\ valid\ (u, t) \equiv ((u[Number] = Num_t) \wedge Maximal(u,t,Num)) \}$$

The following is a slightly more complex example.

```
retrieve (E.Name, MinSalary=Min(E.Salary by E.Dept))
```

Taking it one step at a time, the non-temporal version appears on page 46. The first (incorrect) temporal version is

$$MS_t = \{ m^{(2)} \mid (\exists E)\ ((Employee(E) \wedge valid\ (E, t) \wedge E[Dept] = m[1])$$

$$\supset m[2] \leq E[Salary])$$

$$\wedge\ (\exists F)\ (Employee(F) \wedge valid(F, t) \wedge F[Dept] = m[1]$$

$$\wedge\ m[2] = F[Salary])\}$$

$$\{ u^{(2+2)} \mid (\forall\ t)\ (valid(u, t) \supset (\exists E)(\ \exists m)\ (Employee(E) \wedge valid(E, t) \wedge MS_t\ (m)$$

$$\wedge\ m[1] = E[Dept] \wedge u[Name] = E[Name] \wedge u[MinSalary] = m[2]))$$

}

Again, the period of time u has been valid has been incompletely specified. To specify the maximal period, first define

$$V_t(d) \triangleq v \text{ such that } (\exists E)(\exists m)(Employee(E) \wedge valid(E,t) \wedge MS_t(m)$$
$$\wedge\, m[1] = E[Dept] \wedge v = E[Name] \wedge m[2] = d$$

The correct temporal semantics is then

$$\{u^{(2+2)} \mid (\forall t)\ valid(u,t) \equiv (u[MinSalary] = V_t(u[Name]))$$
$$\wedge\, Maximal(u,t,V(u[Name]))\ \}$$

The semantics of the cumulative CountC operator (version 1) looks quite similar to that of the instantaneous Count operator:

retrieve (Number = CountC(E.Name where E.Dept = "Toy"))

$\Rightarrow NumC_t$ = cardinality of

$$\{\ n^{(1+2)} \mid (\exists t')\ (\exists E)(Employee(E) \wedge t' \leq t \wedge valid(E,t')$$
$$\wedge\, E[Dept] = \text{"Toy"} \wedge n[1] = E[Name])\ \}$$

$$\{\ u^{(1+2)} \mid (\exists t)\ valid(u,t) \equiv ((u[Number] = NumC_t) \wedge Maximal(u,t,NumC))\ \}$$

In English, this says that the cardinality, at any instant. is equal to the number of employees working in the toy department at any time *prior* to the instant in question. The reader may find it useful to compare these statements with their nontemporal counterparts. Note that the function $NumC_t$ does not check for contiguous periods; it counts 2 contiguous periods as two periods, rather than as one.

The semantics of the avgc operator is more complex, since the duration of the period is involved. For the query which determines the average salary by department over time,

```
retrieve (E.Dept, AS=avgc(E.Salary by E.Dept))
```

$$\Rightarrow \{u^{(2+2)} \mid (\forall t)\ (valid(u,t) \equiv (u[AS] = A(u[Dept], t)))\}$$

$$\text{where } A(T,d) \triangleq \frac{1}{T} \int_0^T \sum_{E \in Employee} V(E, (valid(E,t) \wedge E[Dept] = d))dt$$

and $V(E, P, d) \triangleq E[Salary]$ if P and 0 otherwise

There are several assumptions being made in the above statement. In addition to assuming that time is metric (isomorphic to the real numbers) it is assumed that time is *equal tempered,*

that is, the "distance" from t_1 to $t_1 + \Delta t$ is equivalent to that from t_2 to $t_2 + \Delta t$. Otherwise the average is meaningless. Also, it is assumed that there is a designated time $t = 0$. This time is arbitrary, but has a great impact on the value returned by the aggregate. $t = 0$ will usually correspond to the beginning of the experiment. Note that the semantics are independent of the unit of time chosen; the same values will result whether time is measured in microseconds, minutes, or years.

At this point, a formal semantics for the entire TQuel retrieve statement has been presented. However, this semantics has totally ignored the effects of incomplete information. The remainder of this chapter will examine the sources of incomplete information in the process of monitoring, and will extend the semantics to include this indeterminacy.

4.5. Indeterminacy

The development of the semantics presented above assumed complete and accurate information in the relations being used to derive new relations. Unfortunately, there are many sources of incomplete and incorrect information. Collecting all the events concerning a relation may be impossible, due to inadequate processing or bandwidth resources, or the inability to generate the correct types of events. Sampling is another source of incomplete information. It is impossible to sample a relation continuously, and the slower the sampling frequency, the more changes in the relation are overlooked. Finally, the clocks in a distributed systems cannot be totally synchronized, introducing further uncertainty (see section 5.5.2).

The monitor must be able to deal in some way with these limitations, and to determine how valid the collected information is. If it is impossible to acquire the necessary events concerning the primitive relations being monitored, then the monitor must revert to sampling at least a subset of the relations. If the sampling frequency is excessive, then it must be lowered. Each step represents a decrease in the precision of the information available to the user. Such limitations will always be imposed on a monitoring system; it is important that the system can gracefully handle varying amounts of monitoring information, making as much use of this limited information as possible.

A second problem relating to incomplete information is the loss or delay of samples, and the loss of event information. The ramifications of missing or delayed samples should be confined to (a) an increased probability that a condition became true after the last sample and was subsequently invalidated before the next sample, and (b) a certain amount of 'definite' knowledge reduced to 'possible' knowledge.

The problem of lost events (which were generated but never recorded by the monitor) is a more serious one. Probably the only way to successfully deal with this problem is to provide the monitor with the knowledge necessary to detect inconsistencies in the data being col-

lected and (a) generate the missing events, with an appropriate occurrence uncertainty, or (b) at least revise previous information to account for the increased uncertainty. The mechanism presented below is adequate in the presence of samples; handling lost events is beyond the scope of this thesis.

To cope with the presence of incomplete information, all information is classified as *determinant* (i.e., true), *indeterminant*, or false. The *closed world interpretation* (c.f., [Reiter 78]) is adopted, so the statement that the truth value of an atomic formula is false is represented by the absence of a tuple from the appropriate relation. Hence, only determinant or indeterminate information need be explicitly represented.

Events *as measured* are not instantaneous, but are associated with a with a "fuzzy" period specifying when the event might have actually occurred [Kahn&Gorry 75] (see Figure 4-2). Periods are composed of three underlying periods: the time in which they are possibly valid (the *initial* portion), the time they are definitely valid (the *definite* portion), and the time they are possibly invalid (the *final* portion). These uncertainty components are included in the calculation of derived periods and events by the monitor, and are delimited by the designated times. Spacial fuzziness is handled in a similar manner in [McDermott 80].

4.5.1. Semantics

In TQuel, indeterminacy is handled automatically during the execution of the query. There are no special constructs provided for the user to specify how incomplete information is to be processed; instead, existing constructs are associated with semantics describing how an arbitrary degree of indeterminacy is dealt with. In the limiting case, that of no indeterminacy, the semantics should be identical to those just presented. The primary goal in the specification of the semantics is to preserve as much information as possible in the derived relations, while ensuring that such an equivalence is true.

Since the representation of events and periods has just been modified from that assumed earlier, the semantics of the retrieve statement must also be changed. Instead of one implicit domain for events, starttime, there are now two (see Figure4-2): start-indeterminant (or istart) and start-determinant (or dstart). Similarly, the two implicit domains for periods, starttime and stoptime, are replaced by four: istart, dstart, dstop, and istop. Metric time is still assumed; in particular, the following relationship must hold for all event and period tuples:

$$\text{istart} \leq \text{dstart} \leq \text{dstop} \leq \text{istop}$$

(the four implicit domains are each real-valued timestamps). The starttime and stoptime domains appeared in four places in the tuple calculus semantics presented above: in the definition of Order, in the term associated with the start clause, in the term associated with the stop clause, and in the terms associated with the aggregate operators. Each occurrence must be rewritten using the new implicit domains.

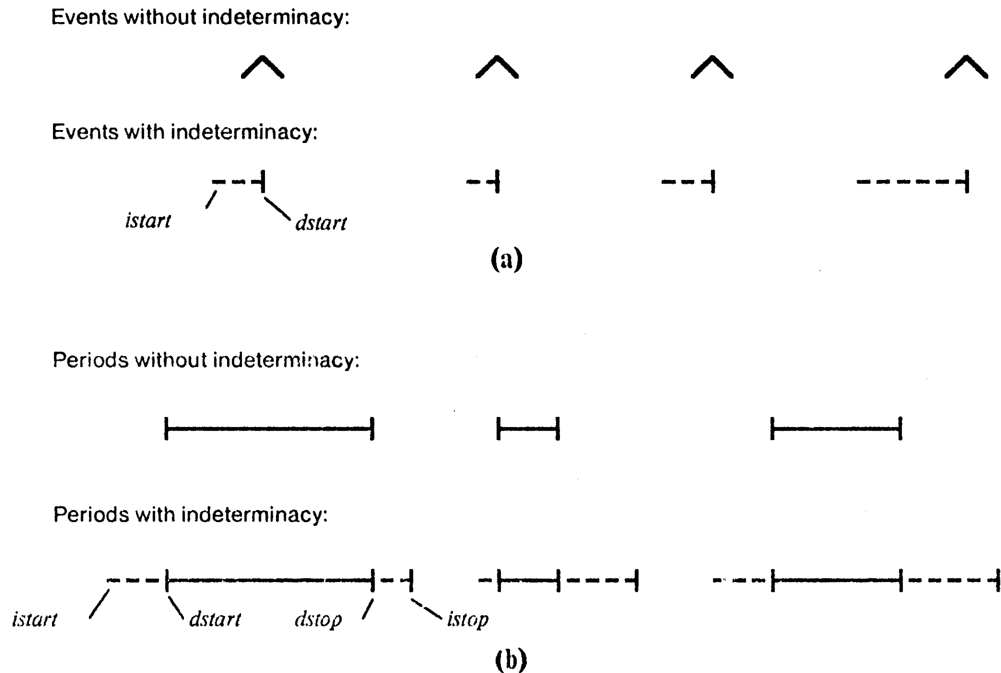

Figure 4-2: Representing Uncertainty

The Order function defines a temporal ordering on the events provided as arguments. This temporal ordering is tested for membership in a set of execution histories derived from the temporal expression in the when clause. However, when indeterminacy is considered, a strict temporal ordering of events may not be possible, due to the overlap of the indeterminant portion of two events or periods. The solution is to instead test each execution history against the argument events, with three possible outcomes: (1) no execution history was satisfied by the given events, (2) one or more execution histories were satisfied, or (3) one or more execution histories may have been satisfied, but the indeterminacy of the events prevents a definite decision. Since each execution history is a sequence of events, these tests involve repeated application of the predicate **before**:

before (a, b) $\triangleq$ a[starttime] $<$ b[starttime]

As an example, the temporal expression

(A ; B) , C

is translated into the following set of execution histories

{ ABC, ACB, CAB }

which may easily be translated into the propositional calculus statement

$$(\textbf{before}(A, B) \land \textbf{before}(B, C)) \lor (\textbf{before}(A, C) \land \textbf{before}(C, B))$$

$$\lor (\textbf{before}(C, A) \land \textbf{before}(A, B))$$

Since indeterminacy requires this predicate to have three results, true, false, and indeterminant (Ω), the tuple calculus must be a three-valued logic system [Rescher 68]. We will base this system upon the following truth tables (where α and β are arbitrary truth values)[11]:

$F \land \alpha = F$	$\alpha \land F = F$	$T \land T = T$	otherwise $\alpha \land \beta = \Omega$
$T \lor \alpha = T$	$\alpha \lor T = T$	$F \lor F = F$	otherwise $\alpha \lor \beta = \Omega$
$\neg F = T$	$\neg \Omega = \Omega$	$\neg T = F$	

The existential and universal quantifiers behave like iterated $\lor$and $\land$, respectively. In the tuple calculus statement,

$$\{ u \mid \alpha \}$$

if α has a truth value of true, then u is in the set; if α has a truth value of false, then u is not in the set (the closed world interpretation); and if α has the truth value Ω, then u is in the set but is indeterminant.

The **before** predicate has two parameters, both events, and is defined as follows:

before (a, b) $\triangleq$ T if a[dstart] < b[istart], F if a[istart] $\geq$ b[dstart], and Ω otherwise

Hence, before(a, b) is true if the entire period when a may be valid occurs prior to any time when b might have occurred, false if b occurs completely before a might have occurred, and indeterminant otherwise. To complete the details, let τ' be as defined previously, and $W'(\tau')$ be the propositional calculus expression derived from $W(\tau')$ as described above, translating a set of execution histories into a disjunction of conjunctions. Define

---

[11] Many systems of 3-valued logic have been proposed. The system adopted here was introduced in 1938 by S.C. Kleene [Kleene 38]. In this system, a proposition is to bear the third truth value Ω if the proposition is unknown. This characterization is in contrast to other systems, such as the one proposed by Lukasiewicz, where Ω is considered somewhere between true and false. Lukasiewicz's system was motivated in part by concerns of "future contingency," whose occurrence, such as that of a sea battle tomorrow (Aristotle's example) is not determinable, especially given free will. Future contingencies are handled in a more elegant fashion with temporal logic [Rescher 71]. Although time appears throughout in the semantics presented here, the application of temporal logic is not necessary, since the formulae only involve temporally definite statements about the past. The primary operational difference between the two 3-valued systems lies in the truth value form $\Omega \supset \Omega$. Kleene's system defines $\alpha \supset \beta$ as $\neg\alpha \lor \beta$, so $\Omega \supset \Omega \rightarrow \Omega$. In Lukasiewicz's system, $\alpha \supset \alpha$ is a tautology, so $\Omega \supset \Omega \rightarrow T$.

$T_1(\delta_i) \triangleq \tau_i$[istart] if $i \leq k$, and τ_i[dstop] otherwise

$T_2(\delta_i) \triangleq \tau_i$ [dstart] if $i \leq k$, and τ_i [istop] otherwise

T_1 defines the start of the indefinite portion of the event in question, and T_2 defines the end of the indefinite portion. Using there functions, before can be defined:

before $(\delta_i, \delta_j) \triangleq$ T if $T_2(\delta_i) < T_1(\delta_j)$, F if $T_1(\delta_i) \geq T_2(\delta_j)$, and Ω otherwise

That is, an event is **before** a second event if it is certain to have occurred before the second event might have occurred. An event is not **before** a second event if the latter event definitely occurred before the former event might have occurred.

To account for indeterminacy, rewrite

$\wedge$ Order(τ') $\in$W(τ')

as

$\wedge$ W′(τ')

Since the tuple calculus used here is a three-valued logic system, W′(τ) may also have a value of Ω. In that case, there is no definite portion in the derived period.

The terms associated with the start and stop clauses are not hard to generalize. Since the function Φ now returns a pair of timestamps (istart, dstart) rather than an individual timestamp, each term is rewritten as two terms:

$\wedge$ u[istart] = Φ_ν $(t_{k_1}, \cdots, t_{k_p})$[istart]

$\wedge$ u[dstart] = Φ_ν $(t_{k_1}, \cdots, t_{k_p})$[dstart]

$\wedge$ u[dstop] = Φ_χ $(t_{m_1}, \cdots, t_{m_q})$[istart]

$\wedge$ u[istop] = Φ_χ $(t_{m_1}, \cdots, t_{m_q})$[dstart]

Of course, the functions contained in Φ will be slightly different. For instance, in the function

Start: $E^2 \rightarrow E$

E ranges over pairs of timestamps (real numbers) instead of individual timestamps.

4.5.2. Defaults

The semantics should also specify the defaults assumed in the language. The defaults for the additional clauses in TQuel should be natural to the user. If only one tuple variable (say, A) is used, and it is associated with a period relation, then the defaults are

```
start A.start
stop A.stop
```

These defaults say that the result tuple is to start when the underlying tuple started and stop when the underlying tuple stopped. When an event relation is associated with the one tuple variable, the default is

```
at A
```

specifying simply that the result tuple was valid at the same instant the underlying tuple was valid.

When two or more tuple variables are used, the situation is more complex. Let us assume initially that all the tuple variables are associated with period relations. The retrieve statement with defaulted temporal constructs looks identical to a standard Quel retrieve statement; thus it should have an identical semantics. An Ingres database is not temporal; instead, it advances in discrete jumps. Whenever a relation is updated, the "clock" advances, and the database is assumed consistent at the new time. Hence, the tuples participating in a retrieval are all valid at the time the query is executed. Extending this semantics to a temporal database is now straightforward: the result tuple is valid at all the points in time when *all* the underlying tuples were valid. Thus, if the tuple variables t_1, t_2, . . . ,t_k are involved in the query, then the default temporal clauses are

```
when (t1.start,....tk.start).stop ; (t1.stop,...,tk.stop).start
start (t1.start,...,tk.start).stop
stop (t1.stop,...,tk.stop).start
```

The start clause specifies that the result tuple is to start the instant all the underlying tuples are valid; the stop clause specifies that the result tuple is to end as soon as any underlying tuple is no longer valid. The when clause states that all the tuples should be valid for a finite period of time, and is equivalent to

```
when (t1 ,..., tk)
```

which indicates that all the tuple overlap each other. If a particular tuple variable t_i is associated with an event relation, simply replace 't_i.start' and 't_i . stop' in the above clauses with 't_i.time', or simply, 't_i'.

When aggregate operators are used in period relations, the decision needs to be made whether to consider the instantaneous or cumulative version to be the default. An argument similar to the one above concerning multiple tuple variables concludes that the instantaneous version more closely preserves the semantics of standard Quel. Hence the **Count** operator will be the instantaneous version; **CountC** must be used if the cumulative version is desired.

4.5.3. Indeterminacy and Aggregate Operators

As before, to formally define the semantics of aggregate operators given indeterminacy, the informal semantics must first be understood. Without indeterminacy, the aggregate operator partitioned the tuples valid at any instant (or valid at any time prior to the instant, for cumulative aggregate operators) into groups, and assigned a value to the tuples in each group. Due to tuples which start and stop at indeterminate times, the resulting value is also indeterminate. Hence, it is impossible to assign a single value to each tuple (see Figure 4-3a).

There are three possible strategies in dealing with this anomaly. The first is the easiest and least satisfying: restrict the value of **Count** to be an integer or the special value *undefined* at any point in time. The value will vacillate between a determinate integer and *undefined* Of course, there still won't be a single valued count for each tuple, but at least there will be a value for each instant of time (see Figure 4-3b).

The second strategy is only slightly more appealing: for those periods when the first strategy assigned *undefined,* provide a value which makes **Count** well-behaved, for an appropriate characterization of "well-behaved" (see Figure 4-3c), and make the tuple completely indeterminant (i.e., no definite portion). This allows **Count** to have an integer value over all time, but is somewhat arbitrary as to the value assigned to indeterminant portions.

The third strategy is to contend with multiple values of **Count** (see Figure 4-3d). On the one hand, this strategy keeps the most information, which might be useful in further derivations. On the other hand, it suffers in at least two aspects. Implementation is much more difficult because an instant might be represented by several indeterminate tuples, with different values. Also, the information is difficult to deal with: is it helpful to know that the **Count** of something at 3 pm was perhaps 3, or maybe 5, or even 16?

The first strategy corresponds to extending the domains with *null values* [Vassiliou 79, Lipski 79, Codd 79]. There are many possible semantics for null values; the one relevant here is "value unknown". The second strategy assigns the truth value Ω to the entire tuple. The third strategy implies a many valued logic, with its inherent complexity. Since the second strategy is most consistent with the representation of periods described previously, it was selected for the initial version of the system.

At this point, there exists an asymmetry in the expressive power of the language. The at clause allows an event relation to be derived from period relations, and the start and stop clauses allow a period relation to be derived from other period and event relations. However, there is no way to derive a period relation from a single event relation. One use of this functionality is the conversion from traced events to the corresponding periods. In this case, the delimiting events are available, and the user specifies that the period be derived. Another use is the conversion from sampled events to the corresponding events. These conversions

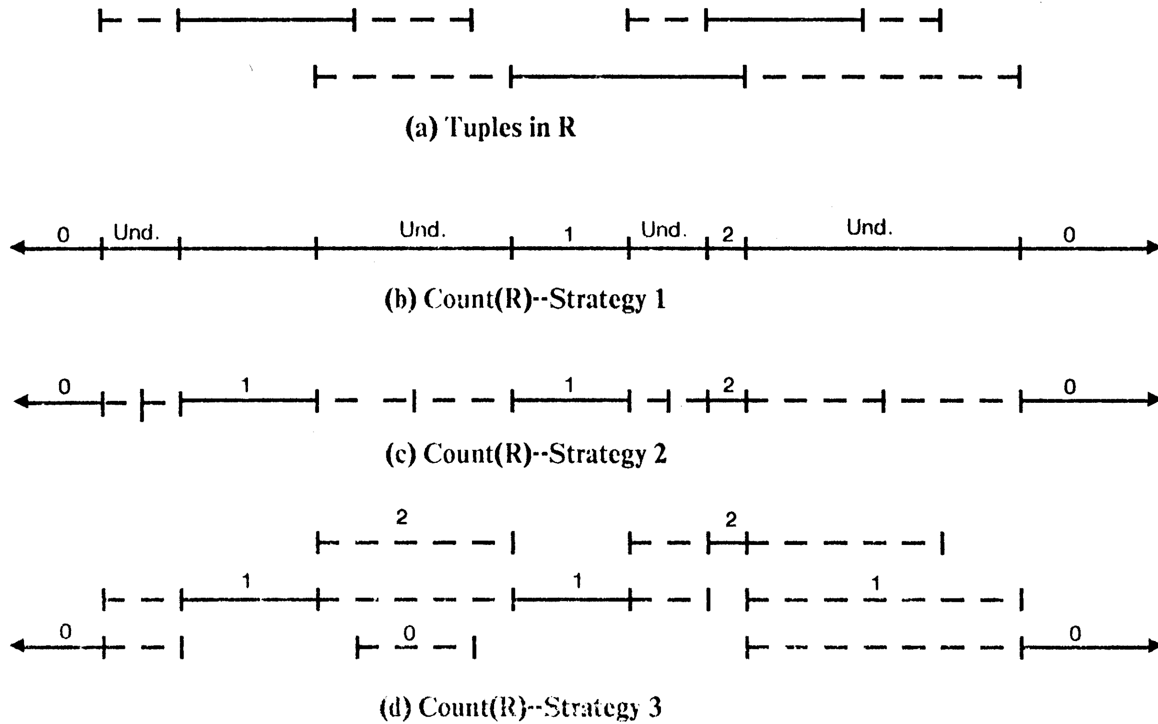

Figure 4-3: Different Strategies for Handling Indeterminacy with Aggregate Operators

differ only in the indeterminacy of the resulting periods. In the case of traced events, the indeterminacy is limited to the indeterminacy in the underlying events; in the case of sampled events, exactly the opposite is true: the *determinacy* is limited to that of the underlying events.

The appropriate conversion operator depends on whether the underlying relation is sampled or traced. There is also the issue of converting a derived event relation into a period relation. Determining whether a derived relation may be considered "traced" for the purposes of converting into a period relation requires substantial knowledge of the functional dependencies of the domains in the relation. Instead, an alternative was adopted requiring less sophistication in the monitor. Traced events, which the monitor knows are traced, are immediately converted to periods. To obtain the underlying events, the at clause may be used. Only one conversion operator is provided for the user; **ExtendC** assumes the underlying events are sampled. This assumption is correct in the best case; in the worst case the resulting relation will be completely indeterminant.

The formal semantics is now straightforward. The only place indeterminacy appears in the semantics of aggregate operators is in the definition of valid, which may be defined using the new implicit domains as

$$\text{valid}(u, t) \triangleq T \text{ if } (u[\text{dstart}] < t) \wedge (t < u[\text{dstop}]),\ F \text{ if } (u[\text{istart}] \geq t) \wedge (t \geq u[\text{istop}]),$$
$$\text{and } \Omega \text{ otherwise}$$

As with $W'(\tau)$, if the truth value for valid is Ω for some time t, then the tuple calculus statement reduces to

$$\{ u \mid \Omega \}$$

for the time t, in which case u is indeterminant at time t.

It remains to be shown that, in the absence of indeterminacy, the semantics just presented is equivalent to that discussed in the first part of this chapter. Clearly, without indeterminacy, the initial and final portions are absent and the following holds for all event and period tuples:

$$\text{istart} = \text{dstart and dstop} = \text{istop}$$

In this limiting case,

$$T_1(\delta_i) = T_2(\delta_i)$$

and thus the before and valid predicates may only have the truth values true and false. Ω is eliminated as a truth value, reducing everything to the standard two-valued logic system, with an identical semantics to that defined initially.

4.6. Summary

This chapter has presented a complete formal semantics for the entire TQuel retrieve statement. The chapter proceeded in an incremental fashion, starting with the basic Quel semantics, then adding the new TQuel clauses and additional semantics for aggregate operators, and finally, adding semantics for indeterminacy. After a short review of tuple calculus and a discussion of the application of path expressions in TQuel, the semantics of event expressions was described as functions on events or pairs of events (periods), ultimately yielding an event. A transformation system provides the semantics of temporal expressions, yielding a set of execution histories on the tuples participating in the expression. At that point, a tuple calculus expression for TQuel retrieve statements without aggregates was presented.

The semantics of Quel aggregates involved an auxiliary relation which was then used in the primary tuple calculus statement. Time was added by using a predicate indicating when a tuple was valid. Indeterminacy involved several changes: use of 3-valued logic, a more complex definition of the before and valid predicates, and a slightly altered semantics for the when clause.

As a result, the semantics of a TQuel retrieve statement is defined in terms of a tuple calculus statement which can be mechanically produced from the TQuel statement. Summarizing,

> **Result:** A formal semantics may be defined for the entire TQuel retrieve statement. This semantics has the following properties: (a) it reduces to the standard Quel semantics when the time domain is fixed at a particular time; (b) it includes aggregate operators in a uniform fashion; and (c) it accommodates an arbitrary degree of indeterminacy.

III. Realization

To verify the thesis that the relational model is an appropriate formalization of the information processed by a distributed monitor, it is necessary to show that the monitor can take a query in a temporal query language (in this case, TQuel) and subsequently gather the correct low level information, process it as directed by the query, and present the high level information to the user, all in a relatively efficient manner. This part describes in some detail the mechanisms and techniques enabling the monitor to accomplish these objectives.

Chapter 5
A Low Level Data Collection Mechanism

Data collection techniques have been at the center of attention in previous work in monitoring, to the exclusion of other areas such as data representation and manipulation. Most papers on the monitoring of user programs are variations on the technique of *profiling* in a variety of programming languages. This approach involves execution counts or timing at the procedure, statement, or instruction level, using sampling or tracing. However, there have been few advances since the early 1960's, when sampling and tracing were first introduced [Plattner&Nievergelt 81]. Data collection for monitoring of operating systems has relied on sampling or tracing [Nutt 79]. Techniques for using special hardware have been more innovative: since additional logic imposes no overhead on the computation, capabilities such as event counters, combinational and sequential logic on events, comparators, and histogram generators can be provided [Bonner 69, Wulf *et al.* 81]. Network data collection has concentrated on performance evaluation issues and has, in general, been confined to techniques mentioned above [Abrams&Treu 77, Nutt 79].

Recent systems have taken a more integrated approach to monitoring, attempting to reduce the great effort necessary when using the low-level tools previously available. A unified set of facilities for monitoring a packet radio network was developed at UCLA [Tobagi *et al.* 76]. Gertner's system [Gertner 80], described earlier in section 1.4, allowed relatively painless monitoring of a distributed system at the message passing level. The Computer Network Monitoring System (CNMS), a rather ambitious system designed at the University of Waterloo, used a sophisticated combination of hardware and software to monitor a geographically distributed network [Morgan *et al.* 75].

In spite of these developments, an integrated approach to monitoring data collection is still lacking. One possible strategy for developing such an approach is to start with the relational model described in the previous chapter. Unfortunately, the relational model is too general to be of much help in characterizing the data to be collected. Another possible strategy proceeds by developing a conceptual model of the behavior of the program to be monitored, and attempts to represent that behavior within the relational model. This strategy will be the one pursued in the present chapter. The next section begins with a comparison of data collection as performed by a conventional data base system and by a monitor. A model of the environment where the data collection takes place is then presented, followed by a discussion

of the properties an effective mechanism must have. The remainder of this chapter will present such a mechanism and examine how various aspects of the environment impact it. The discussion will be independent of any particular operating system; details of the implementation of the mechanism in two operating systems can be found in chapter 8. However, it is assumed that the monitor is partitioned into two communicating components: a *resident* portion, performing those functions requiring close interaction with the monitored system, and a *remote* portion, performing the functions requiring close interaction with the user. Both portions together comprise the complete monitor. This separation is necessary when monitoring a distributed system, where a resident monitor would exist at each processor, sending collected data to the centralized remote monitor, which may or may not execute on one of the processors being monitored. Functionally, the resident monitor collects the event records and interacts with the operating system, and the remote monitor analyzes and displays the monitoring data. These functions will be discussed in more detail in Chapter 8.

5.1. The Environment

Data collection in the monitoring domain differs from that in conventional database systems in several ways. In most information processing systems, the emphasis is on information manipulation and retrieval, with minimal aids for data collection. Although some systems provide tools for key entry and point-of-sale data acquisition, data collection remains difficult to automate, because the highly-structured databases must interface with much less regimented mechanisms: written and oral communication, multiple incompatible data representations, psychological and societal constraints. The monitor, as an information processing system, has much more control over the collection of data, since that data is already available in digitized form, either resident on a bus line or network link, or stored in registers, main memory, or on disk, certainly in a more convenient format.

The availability of monitoring information results in a second distinction between data collection as performed by conventional database systems and by the monitor: the monitor in general must contend with massive quantities of data, only a small portion of which may be useful to the user. For example, suppose that the monitor receives a value-time pair for each change in the program counter. The monitor would have to run on a machine several orders of magnitude faster than the one being monitored merely to store the information. However, if only routine timings were desired, the grain of data will be much coarser. As another example, suppose that timings were desired only for a single routine. Unless the data collection mechanism supports *filtering*, where only data satisfying specified constraints is actually collected, the monitor will have to contend with data concerning all routines. Extraneous data is expensive, because computing resources are required to collect it and to decide to discard it. Thus, data collection for monitoring involves careful selection of data, rather than access and conversion to a more useful representation, as in conventional information processing systems.

In order to discuss the data collection mechanisms, it is necessary to characterize the environment in which the mechanism executes. The model employed here has been used in several recent operating systems [Wulf *et al.* 81, Jones *et al.* 78] and languages [Ichbiah *et al.* 79, Shaw 81], although the model can be used to conceptualize program behavior in any system [Jones 77]. The environment is defined to be a collection of *strongly typed objects*, both passive (e.g., data structures) and active (e.g., processes). *Type managers* export functions to be applied to objects of the type(s) supported by the manager; all operations on an object are performed by the type manager through well-defined interfaces (implying the existence of a type-checking mechanism). This model thus identifies the *operation* being performed on the *object* by the *performer* (the type manager) as a result of a request by an *initiator*. The user can create new types by defining the representation of the object and specifying the operations which can be performed on objects of that type. The model applies to all levels of granularity; in particular, a type manager may be implemented in hardware, firmware, or software.

Examples of type managers include an encapsulation of a set of routines (c.f., Ada packages [Ichbiah *et al.* 79], or a task force [Jones&Schwans 79] or even a data type and supporting procedures as found in Pascal [Wirth 71]). The more general term used here, type manager, avoids limiting the model to a particular language or operating system. The model is especially applicable to monitoring because it forces state changes to be precisely specified: any change to the representation of a data structure (i.e., object) must occur within a function of the type manager as a result of performing a defined operation. Control flow can also be characterized in this manner: all changes in the execution state of an active object (a process) can be accounted for by examining the sequence of operations performed by the process.

There are several properties which should be satisfied by the data collection mechanism. The mechanism should support strong typing, in that typing violations are not necessary to perform the data collection. Multiple type managers should be permitted. The mechanism should rely as little as possible on cooperation by the type managers, to allow additional data types and type managers to be monitored easily. The mechanism should be efficient, especially when disabled. The mechanism should be selective, allowing the monitor to specify exactly the information to be collected, thereby supporting filtering. The latter two properties allow many sensors to be included in a task force. Both sampling and tracing should be possible using the same mechanism. It should be adaptable to different monitoring granularities. And finally, the mechanism should exhibit good software engineering.

5.2. The Mechanism

The data collection mechanism employed in the monitor relies on the type model presented earlier. An *event* occurs in the context of an operation as defined in the type model. Information concerning an event is collected into an *event record*, which is potentially of variable length and which contains both predefined and user-defined fields. Event records are typed, with each event type being produced by a particular *sensor*. The sensor collects the relevant information concerning that event, and sends the information to the remote monitor. Each sensor is placed in a type manager, and is associated with an operation (or set of operations) provided by the type manager. For example, the file system (a type manager for the file object type) may have a ReadFile event sensor located in the code performing the read operation. Other sensors, such as OpenFile, ExtendFile, PhysicalBlockRead, and ModifyProtection, may also be present in the file system. Since state changes on a file object can only occur as the result of operations performed by the file system, sensors within the file system can monitor these state changes for all file objects.

Event records always contain the name of the type manager performing the operation (if the type manager is itself an object), the event type, the name of the referenced object, and the name of the initiator. Thus, all four components of a state change are recorded for later analysis. Additional information, including the time the event occurred, may also be recorded in the event record.

Event records are stored in *receptacles*, which handle the enabling and synchronization of event records. Receptacles constitute a bridge between a user-defined type and the monitor: the sensors residing in the type manager for this type use the receptacle to communicate event records to the monitor. Similarly, the monitor uses the receptacles to enable or disable sensors in the type manager. Receptacles are abstract objects in their own right, whose type manager is the resident monitor. Events are *enabled* by the resident monitor by setting switches in the receptacle. Locks in the receptacle arbitrate simultaneous access and modification of the switches by the resident monitor and the sensors.

Figure 5-1 shows a portion of two type managers (written in pseudo-Pascal with type extensions). There are two sensors shown, both in the WriteBlock operation. There are several operations on receptacles supported by the resident monitor. To install a receptacle in an object, the type manager for that object requests a receptacle from the resident monitor. The type manager then places the receptacle in the object at a location determined by the type manager. To enable (or disable) an event for an object, the resident monitor presents the object to the appropriate type manager with a request for the receptacle contained in the object. The resident monitor then modifies the appropriate switch in the receptacle. Therefore, the minimum functionality a type manager must provide to support monitoring is the *access receptacle* operation and the *install receptacle* operation. Both of these are straightforward to implement (see section 8.3).

```
TypeManager FileSystem =

          Requires Monitor;
          Exports Type File = Record
                                ...
                                R: Receptacle;
                                ...
                                End;

Private Var TMReceptacle: Receptacle;

Procedure InstallReceptacle (AFile: File) = AFile.R := NewReceptacle();
Function AccessReceptacle (AFile: File) returns Receptacle = AFile.R;
Procedure WriteBlock (AFile: File; BlockNum: Integer; Contents: Page) =
          ...
          StoreEventRecord (AFile.R, WriteBlockEventA, BlockNum, ...);
          ...
          StoreEventRecord (TMReceptacle, WriteBlockEventB, ...);
          ...
          ;
...
End;

TypeManager Monitor =

Exports Type Receptacle = Record
                    ...
                    End;

Function NewReceptacle returns Receptacle = ...;
Procedure EnableEvent (Object: Any; EventNumber: Integer) =
          AccessReceptacle(Object).Enable[EventNumber] := true;
Procedure DisableEvent (Object: Any; EventNumber: Integer) = ...;
Procedure StoreEventRecord (R: Receptacle; EventNumber: Integer; ...) = ...;
End;
```

Figure 5-1: Skeletons of the Monitor and User Type Managers

Any sensor using the receptacle contained in an object must reside in the type manager for that object. When such a sensor is encountered during the processing of a requested operation, the event identification, object name, initiator name, performer name, and receptacle, as well as any additional information provided by the sensor, is passed to the resident monitor, and a *store event record* operation is performed by the resident monitor. First the appropriate enable switch in the receptacle is checked to ensure that the event is enabled for this receptacle. If so, an event record in the proper format is then sent to the remote monitor.

Placing the enable switches in the receptacle allows great flexibility in the enabling of events. Receptacles are associated with both passive and active objects. A receptacle associated with a passive object arbitrates the collection of monitoring information for that object. Enabling the file read event for the receptacle associated with a particular file causes event record to be collected for file reads *only for this file* by any file system process. On the other hand, enabling the file read event for the receptacle associated with a particular file system process causes event records to be collected for file reads on any file performed *only by this file system process*. In the example in Figure 5-1, WriteBlockEventA is enabled for a particular file by enabling the event in that file's receptacle. To enable WriteBlockB, the receptacle associated with the type manager (TMReceptacle) must be modified.

The placement of the receptacle allows the event records to be filtered along three dimensions: by object, performer, or initiator. Each sensor supports filtering of an event type in only one dimension. However, several sensors (and event types) can be associated with an event occurrence (such as file read), each designating a different receptacle to arbitrate event record generation by the sensor. Viewing the set of possible event records as a discrete four-dimensional space with axes consisting of event types, objects, performers, and initiators, the event records generated by a particular sensor form a two-dimensional plane, parallel to two of the axes, and intersecting the event axis and one other axis. The event records enabled by a particular receptacle form a series of two-dimensional planes, all parallel yet intersecting the event axis at different points[12].

Higher degrees of filtering are also possible. A line in the event space represents enabling events on combinations of two of the components of the operation, such as a file read by *this* file system on *this* file. A point represents total control over which event records get generated: a file read by *this* file system process on *this* file, as requested by *this* initiator. Achieving higher degrees of filtering requires either additional information to be stored in the receptacle, and additional processing to determine if the event is indeed enabled, or new receptacles representing component pairs or component triples to be created and associated with the participating objects. Both alternatives require greater than linear space and/or time in the number of objects, and thus are unacceptable in an environment supporting many objects.

The typing model applies to all levels of abstraction, from the hardware (with objects such as memory locations, interrupt lines, device registers), the firmware, the language level (with objects such as variables, semaphores, procedures), to the process level and the program level (with objects such as servers, databases, users). The data collection mechanism can be used at all of these levels, presenting a consistent interface to the rest of the monitor: event records containing the event type, object, performer, and initiator, as well as other, event-specific, information. By associating receptacles with the objects defined at a particular level of abstraction, the full filtering capabilities can also be realized. However, the implementation will differ from level to level, and there must be ways to transmit the information gathered at the lower levels, especially at the hardware and firmware levels, to the higher levels where they can be dealt with by the monitor. Chapter 8 will describe an implementation at the language/process level.

---

[12] A point in this space may include several event records, each representing the same event, object, performer, and initiator, but occurring at different times. Of course, time could be considered as yet another dimension. Visualizing the event space is more difficult with such a change; the author finds four dimensions hard enough!

5.3. Integrating Sampling and Tracing

In the preceding discussion, the assumption was made that the event record is sensed and communicated to the remote monitor when the event occurs. Such event records are called *traced* event records, since their generation is *synchronous* with the event, and thus with the operation whose object, performer, and initiator is named in the event record.

Sampled event records, on the other hand, are generated at the request of the monitor, *asynchronously* with the event. As an example, a sensor located in the scheduler of an operating system could generate *traced* event records pertaining to context switching: process x started running at time t_1, process y started running at time t_2, etc. Another sensor located in the scheduler could generate *sampled* event records at the request of the monitor: process z is now running.

In the context of the type model, both sensors were executed as a result of an operation supported by the scheduler; the former by the dispatch operation, the latter by the "report current process" operation. The only distinction is the nature of the initiator-- either a random process in the system, or the resident monitor. As far as the low level data collection mechanism is concerned, there is absolutely no difference between sampling and tracing: the sensor, when encountered in the course of executing the operation, checks the enable switch in the appropriate receptacle, and, if set, sends the event record to the remote monitor. Of course, the higher levels must treat sampled event records somewhat differently than traced event records, although section 2.2 presented evidence that these differences are not as fundamental as previously thought.

There are several means by which the resident monitor may request sampled event recods to be generated by the user's type manager. The most obvious method is to have the user provide an entry point for the monitor. The process would then wait to be invoked, either by another user process or by the monitor. Another method is to have the resident monitor simply set an enable flag in the receptacle, and assume that the user process regularly checks this flag. A useful mechanism designed for this case is a *one-time-enable* flag in the receptacle. If this flag is set, the next time a particular event record is generated, the event will be disabled, thereby allosing exactly one event to be detectected. A third technique, sharing aspects with the other two, is to send the user process a message when sampling is desired. Although this technique assumes that the process regularly checks a particular mailbox, the sampled event record could be generated while the user's process is performing another function. All three techniques involve the special code in the user's process, yet most of the added complexity is where it should be: in the monitor rather than in the user's (i.e., type manager's) code.

5.4. Other Uses for Receptacles

Since receptacles are the primary means of communication between the resident monitor and the suer's application processes, it may be desirable to add fields to receptacles for sending information other than event records to the resident monitor. Placing counters in receptacles can eliminate the overhead of generating the event records. Counters are also useful for keeping track of the number of missed events, or for accumulating an estimate of the processing overhead of generating events. Other possible aggregation mechanisms include histograms and higher moment generators. The primary requirement for an aggregation operator implemented in this way is that the state change of the aggregate function (such as a sum) can only depend on the current event (e.g., the value to be added to the sum) and the current state of the aggregate (e.g., the current sum).

Counts and sums can be quite useful in themselves. The values of the counters at a particular point in time as well as the behavior of the values over time can be very informative. Most raw measurements on Cm* in the past have consisted purely of counts and sums [Jones&Gehringer 80, Dannenberg 81].

5.5. Interaction with the Remote Monitor

Since the remote monitor contains a large knowledge base concerning monitoring, it is important that it be system independent. If the remote monitor had to be designed and implemented from scratch for each operating system, then much of the effort would be expended getting a minimal set of functions working correctly, rather than enlarging a common knowledge base. Hence, the remote monitor's view of the world is an abstraction supported by the resident monitor interacting with a particular operating system with certain assumptions being made about the event records being generated. These assumptions may be easy to satisfy in one resident monitor, yet difficult to satisfy in a different resident monitor, where the operating system supports rather different functions. There is a tradeoff between strong assumptions, which are difficult to support by the resident monitor, and weak assumptions, which make the derivation of high level information from event records difficult. This section is concerned with the environment as seen by the remote monitor, and how this view is supported.

The format of the event record is one aspect to be standardized. Each event record is divided into a fixed and a variable part. The fixed part is identical in format in all event records, and included the event number, the (possibly nil) names of the initiator, performer, and object, and (possibly) a timestamp. The variable part contains the values from domains of the event record, i.e., the additional information provided by the sensor. Domains must be formatted in a defined external type (currently either a short integer, a long integer, a variable length character string, or a remote name; see section 5.5.1), with the sensor mapping values

in an internal format to an external type. Names and timestamps are much harder to standardize; the rest of this section will describe our approach to these two issues in the context of data collection.

5.5.1. Naming

Who hath not owned, with rapture-smitten frame,
The power of grace, the magic of a name?

--*Thomas Campbell*, in Pleasures of Hope

There are several name spaces within the monitor for objects supported by the operating system; this section is concerned with *internal names* (the operating system specific names), *remote names* (system independent names which are processed by the remote monitor), and the mapping between these two name spaces. Other chapters will deal with the remaining name spaces active in the monitor.

Internal names allow the resident monitor to gain access to the object in question. The internal name for a file may be the disk address of its directory entry, or the *inode* number in the case of Unix files [Ritchie&Thompson 74]; the internal name for a process might be a memory address of a process control block, or an offset into the process table. Object-based systems allow a consistent naming scheme to be used; the operating system supports the addressing of *all* objects by using a name in a standard format. These names are usually protected, so that processes must acquire names, rather than being allowed to arbitrarily generate them. StarOS [Jones *et al.* 78] and Medusa [Ousterhout *et al.* 80], the two operating systems the monitor was implemented on, are both object-based systems; internal names are called capabilities and descriptors, respectively. A remote name is an integer which can be mapped to an internal name, thereby allowing the actual object to be referenced by the resident monitor.

It is helpful to examine briefly how names in these two name spaces are used by the monitor. When the monitor starts, it knows no names. The user issues a query, and the remote monitor instructs the resident monitor to find certain objects and to return the remote names of these objects. At that point, the remote monitor sends some of these names back to the resident monitor, instructing it to enable certain events on those objects. As these events subsequently occur, event records containing the remote names are sent to the remote monitor.

In order for this interaction to occur successfully, several invariants concerning remote names must be guaranteed:

Uniqueness - Remote names must be unique (one remote name for each object) for event records to make any sense at all.

Injectivity Remote names must be unambiguous (one object for each remote name), for the same reason.

Bidirectionality The mapping must be bidirectional; in particular, the resident monitor must be able to find the object given a remote name.

Completeness When an event record is sent to the remote monitor, the remote name for the object, the sensor, and the initiator must be available.

Lifetime The mapping must allow operating system objects to be garbage collected.

Unfortunately, there is no mechanism for producing remote names which will satisfy all five invariants, although different mechanisms violate different invariants. Assume we have a remote name for an object. We can either

- not allow garbage collection, or
- allow garbage collection, and violate the bidirectionality invariant (if the object is deleted, we cannot map the remote name to an internal name), or
- violate the injectivity invariant (map the old remote name onto a new object which has the same internal name as the old, deleted object).

The second and third alternatives apply to operating systems not supporting a consistent, protected internal name space. For example, the disk address of a file may be a valid name for the file while it exists (assuming the directory is not reorganized). However, there is no guarantee that the file will not be deleted and the entry replaced with that of another, newly created file.

The approach adopted here applies to object-based operating systems, and will support capability addressing at the expense of a slightly corrupted bidirectionality invariant: sometimes the mapping from the remote name to the internal name will not work. To see how this is done, we must first explain the mapping between internal and remote names.

Although each internal name at any given time is bound to at most one object, an internal name will in general be bound to a series of objects. Each time an object is garbage collected, the internal name no longer refers to that object, and the time period the binding was in effect, called the *epoch*, is ended. Since each remote name must be associated with a unique object, the remote name will consist of two components, the internal name and a designator of the epoch, to disambiguate the internal name. Remote names will be of the form <epoch><minor name>. The <minor name> will be formed from the internal name in a

system-dependent fashion (this does not circumvent protection, since the internal name may be *read*, but not *written*). The <epoch> for a particular <minor name> is an integer initialized to zero an incremented each time the object associated with the <minor name> is garbage collected. Whenever an event record is constructed, the current <epoch> for each <minor name> in the event record will be concatenated with the <minor name>, thereby forming a remote name.

The resident monitor will maintain a list of internal names it has collected from the event records it has sent to the remote monitor. Whenever a remote name is sent to the resident monitor, the associated internal name will be retrieved using the <minor name> field. Whenever an object is deleted, the <epoch> for the <minor name> for that object be incremented, to ensure that the next object created with this internal name will be mapped to a unique remote name. Note that interaction between the garbage collector and the resident monitor is required to keep the <epoch> values consistent.

The mechanism outlined above satisfies all of the invariants except the garbage collection invariant. As long as an internal name resides in the resident monitor, the object it refers to cannot be garbage collected. Having many internal names in the resident monitor imposes an unnecessary processing and memory burden on the system. Therefore the mechanism must allow internal names to be removed from the resident monitor.

Stated simply, an internal name should be removed if it will never be needed again[13]. Since it is impossible to predict when this will be the case, various approximations may be used:

- the object has an explicit *destroy* operation performed on it;
- there is only one internal name (the resident monitor's) referencing this object (and thus, the object is nonexistent as far as the rest of the environment is concerned);
- the remote monitor will never send the remote name to the resident monitor in a request (or, as a weaker predicate, the remote monitor doesn't have a remote name for this object);
- it has been a long time since an event record has been generated;
- the user has specified that this object is unimportant (or equivalently, has not specified that this object is important);

---

[13] Put another way, all the invariants can be satisfied given an omniscient resident monitor!

- the object is of an unimportant type (or equivalently, is not of an important type).

The first alternative applies only to objects which have a *destroy* operation that can be monitored. The semantics of the destroy operation from the monitor's point of view is that, after the operation has been performed, nothing interesting will ever happen to this object, i.e., no event records containing a name for this object will ever be generated. The second alternative depends on a garbage collector which can determine whether there is only one extant internal name; garbage collection using reference counts would admit this alternative. The third alternative requires garbage collection of names in the remote monitor, effectively extending the capability name space to include the address space of the remote monitor. The last three heuristics depend on psychological aspects, and thus require human factors experiments to determine their effectiveness.

5.5.2. Time

A person with one watch knows what time it is;
a person with two watches is never sure.

-- *Proverb*

Time is a difficult problem when monitoring a distributed system. The event records contain times relative to the local clock of the processor on which the event occurred. In general, the local clocks on physically separate processors will be arbitrarily out of synchronization. Unfortunately, it is theoretically impossible to synchronize imprecise physical clocks over a geographically distributed network with nondeterministic transmission times [Lamport 78]. A more practical constraint is ensuring that the overhead incurred in synchronizing the local clocks remains acceptably low.

There are at least three possible approaches to the local to global time mapping: (1) the mapping is done by a distributed algorithm in the system being monitored; (2) the mapping is done in the remote monitor, using local times sent from the resident monitors; or (3) the remote monitor uses only local time in its calculations. Option (3) was rejected immediately because it would disallow queries involving interactions between processors, which is the reason behind having a distributed monitor in the first place.

Option (1) can be accommodated easily using the Lamport algorithm. The time-keeping algorithm can be embedded in the operating system itself, with timestamps appended to every message, or in the monitor, with timestamps included in messages sent by the monitor. Note that the monitor may be able to adequately maintain a global clock with few additional messages. The most obvious algorithm for option (2) is to simulate Lamport's algorithm in the remote monitor. This approach incurs a greater overhead than Lamport's algorithm itself, due to the additional communication necessary. Another consideration is that if the operating

system provides a reliable communication mechanism, supporting recovery from lost messages or crashed processors, then a global clock is probably already computed by this mechanism (e.g., [Nelson 81]; all reliable communication mechanisms known to the author use some kind of global clock.) In any case, if a global clock is provided by the monitor, other components of the operating system may profit from its presence.

The global clock will of necessity have some error; the maximum skew can be bounded in Lamport's algorithm. Lamport's algorithm also has the property that it preserves the partial ordering of message transmission followed by message reception. The partial ordering necessary for debugging will be preserved and the (unknown) total ordering will embed this partial ordering. The semantics of TQuel explicitly incorporates a skewed global clock.

Given these considerations, we will assume that a global clock is implemented by a distributed algorithm, and is available to each resident monitor. If such a clock is not feasible due to efficiency constraints, as in some real-time systems, then more sophisticated approaches, yet to be developed, are necessary.

5.6. Summary

This chapter first presented a model for the environment the data collection mechanism was to execute in, the type model, and then a mechanism which relies on this model. The occurrence of an event is tied to four components: the operation, the object being operated on, the performer of the operation, and the initiator of the operation. These components are recorded in the event record generated by the sensor, along with additional information germane to the event. A new type of object, the receptacle, was introduced to arbitrate the generation of event records, and great flexibility in filtering was shown to be possible by associating the receptacles with the various entities participating in the event. We demonstrated that, from the point of view of the sensors, there was absolutely no difference between traced and sampled event records, the distinction lying instead in the identity of the initiator of the operation involving the sensor. Finally, several issues involved with the interaction of the remote monitor, in particular naming and time, were discussed, and techniques were developed for coping with the problems inherent in those areas.

At this point, it is possible to answer the query of chapter 2,

> **Problem:** Is it possible to provide effective data collection mechanisms?

with a resounding Yes!:

> **Result:** After examining the type model, a data collection mechanism was designed that supports a high degree of filtering, integrates sampling and tracing, and admits solutions to the problems of naming and time.

Throughout this chapter, we have assumed that any mechanism we can devise to overcome the problems of data collection in a distributed, strongly typed environment could be used effectively by higher levels of the monitor, so that the user is still presented with a simple relational query language interface. This assumption results in an extension of the next problem raised in chapter 2 that is now somewhat harder to solve:

> **Problem:** How can the dynamic incremental updating of temporal relations be implemented effectively, and in such a manner that the facilities supplied by the data collection mechanism are also used effectively?

Chapter 6
The Update Network

As discussed in previous chapters, event records are generated by sensors, collected by the resident monitor, and eventually sent to the remote monitor for further processing. The user specifies the nature and extent of this processing through queries in TQuel, a high level, nonprocedural language. This chapter describes in detail the target of the query translator: an *update network.* The translation of the query into an update network is the topic of the next chapter.

Current relational database systems provide constructs for deriving new relations and for modifying existing relations through the addition, removal, or modification of individual tuples or collections of tuples satisfying some predicate [Ullman 82]. A derived relation may or may not be modified when an underlying relation is modified. If the relation will not be modified (the normal situation), then the derivation is performed once and a new relation is created. If a derived relation is to reflect the changes made to an underlying relation, then a different implementation strategy is used. The relation, called a *view*, is never actually computed. Instead, a query involving a view is modified by the database system to operate directly on the underlying relations, merging the operations specified for deriving the view with the operations specified by the query. Since the underlying relations are accessed whenever a query refers to theview, the data is guaranteed to be current.

Due to the dynamic nature of a temporal database, neither approach is appropriate. Instead of first collecting the primitive relations and then performing queries on them to produce derived relations, the system should process the tuples as they become available, for two reasons: the monitor can perform the collection and computation in parallel, and the derived information can be presented in (somewhat delayed) real-time. The first reasons relates to efficiency; the second one to efficacy.

6.1. Incremental Updating of Temporal Relations

Incremental update algorithms for temporal relations accept information in the form of "this relationship between these entities was true for the period from the time t_1 through t_2", and use this information, plus stored information concerning the relation, to derive an updated relation. Relations evolve in time through tuples (rows of a relation) being added and removed. These changes cause relations derived from a relation to acquire or lose tuples of their own, a process continuing until the new information has been completely assimilated by the relations defined in the system. It is thus natural to concentrate on the flow of tuples (being added and removed) among the relations that are associated with each other through TQuel queries. This viewpoint results in an implementation which is substantially different from those of conventional data base systems.

The monitor will process the tuples using an *update network* produced from the user's query. The update network is specified as a directed acyclic graph (dag). The nodes in this graph are classified as either *access* or *operator nodes*. Information in the form of tuples flows out of the access nodes (which communicate with the resident monitor) and through the network. Operator nodes take tuples from one or more lower nodes and produce tuples which will be sent on to other nodes. The entire network is driven by tuples originating in the access nodes.

It is important to recognize that two radically different paradigms are at work. One paradigm was introduced in chapter 2: the process of monitoring is profitably conceptualized *by the user* as the collection of time-varying relations which can be manipulated by a temporal, non-procedural query language. The paradigm introduced in this chapter has a different scope: the process of monitoring is profitably structured *by the upper levels of the monitor* as the creation and execution of a specialized update network which processes the information collected by the resident monitor.

This network approach emphasizes the flow of information from the sensors (access nodes) to the user. Intermediate relations are not explicitly represented in the network; instead, a relation consists of all the tuples appearing at the output of a particular node while the node is in the network. Operator nodes which store and retrieve long-lived relations, and which display relations, are provided. The instantaneous snapshot of each relation is partially contained in the internal state of the node computing that relation.

The update network provides an implementation of the TQuel retrieve statement, and is therefore procedural by design. The tuple calculus semantics of the Quel (and TQuel) retrieve statement as presented in the previous chapter is non-procedural, also by design. The connection between the two formulations can be seen by considering a third formulation: an algebra on relations. Recall that the tuple calculus statement for the following skeletal Quel statement

```
range of t1 is R1
...
range of tk is Rk
retrieve (ti1 . D1 , ... , tir . Dr)
where Ψ
```

was

$$\{u^r \mid (\exists t_1) \ldots (\exists t_k)(R_1(t_1) \wedge \ldots \wedge R_k(t_k))$$
$$\wedge\, u[1] = t_{i_1}[j_1] \wedge \ldots \wedge u[r] = t_{i_r}[j_r]$$
$$\wedge \Psi')\}$$

(Recall that D_m is the $j_m{}^{th}$ attribute of the relation R_{i_m}). The relational algebraic formulation for the same Quel statement is

$$\pi_\alpha(\sigma_{\Psi''}(R_1 \times R_2 \times \ldots \times R_k)),\ \text{where}\ \alpha = t_{i_1}.D_1, \ldots, t_{i_r}.D_r$$

which forms the cartesian product of the underlying relations $(R_1 \times R_2 \times \ldots \times R_k)$, applies the appropriate selection $(\sigma_{\Psi''})$, and projects the desired domains (π_α). It can be proved that each tuple calculus statement has an equivalent relational algebraic statement [Ullman 82]. In general, however, the relational algebraic statement is much easier to implement.

The relational operators can be augmented to support the features in TQuel. The projection operator may be used to select the implicit time domains as specified by the start, stop, and at clauses. The selection operator may be used to select tuples satisfying the when clause. Hence, the TQuel retrieve statement

```
range of t1 is R1
...
range of tk is Rk
retrieve (ti1.D1 , ... , tir.Dr)
where Ψ
when τ
start υ
stop χ
```

has the associated relational algebraic form

$$\pi_{\alpha'}(\sigma_{\tau'}(\sigma_{\Psi''}(R_1 \times R_2 \times \ldots \times R_k))),\ \text{where}\ \alpha' = \upsilon', \chi', t_{i_1}.D_1, \ldots, t_{i_r}.D_r$$

The parse tree of this expression is shown in Figure 6-1. The update network is a modified version of this tree. The operators correspond to the operator nodes in the network, and the tuple variables associated with the primitive relations correspond to access nodes.

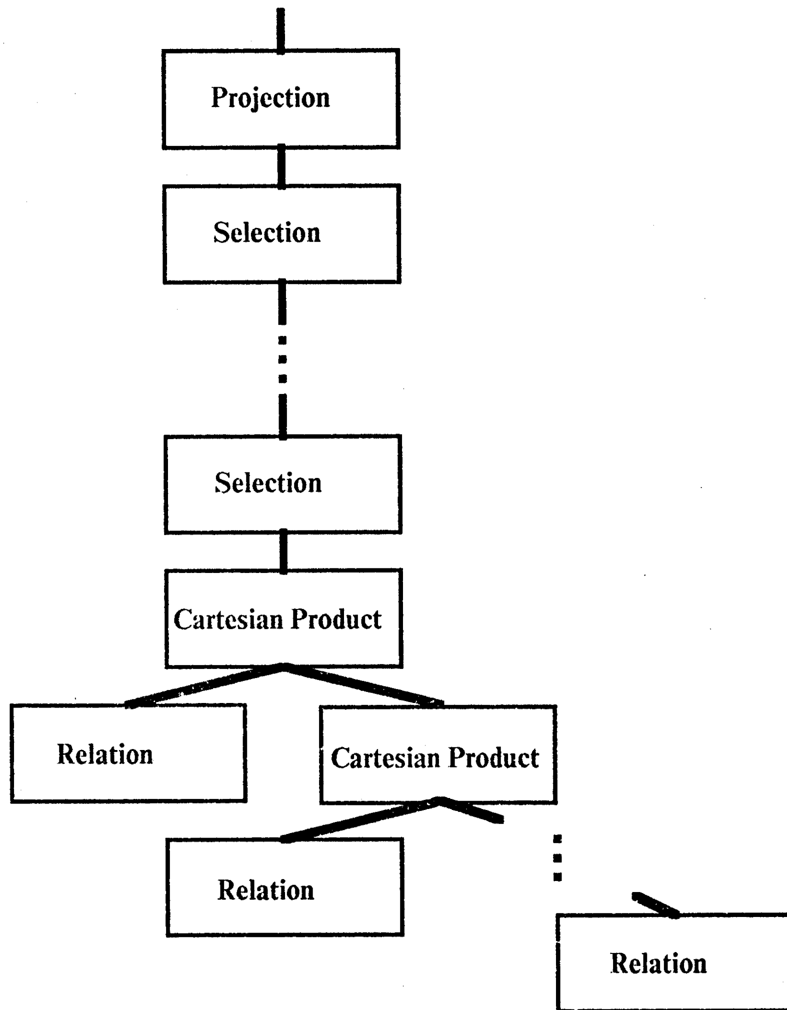

Figure 6-1: Parse tree for the relational algebraic expression for the skeletal TQuel retrieve statement

6.2. Generic Nodes

The access and operator nodes present in the update network are instantiated from a set of predefined *generic access* and *generic operator nodes*. Some of the operator nodes have parameters, equivalent to the subscripts shown in the relational algebra. For instance, the projection operator takes as an argument a list of the domain to be projected.

6.3. Access Nodes

Access nodes are the mechanism by which information collected by the resident monitor enters the update network. Each generic access node is associated with an event type, and thus with the set of sensors generating event records of that type. Access nodes are instantiated from generic access nodes, and are placed in the network. When a sensor generates an event record, the appropriate tuple is placed on the output arc of all access nodes instantiated from the appropriate generic node. At this point, the tuples start flowing through the network and the processing commences.

Since the access node is the counterpart of the sensor in the network, it must have total control over the sensor. In particular, the access node must in some way determine

- which sensors or objects are enabled, and

- when samples are to be taken (if the event is sampled).

A second restriction is that the control functions of the access nodes must interface cleanly with the general paradigm of tuples flowing in the network. Both functions are accommodated using additional input arcs.

The approach taken to control enabling is to add input arcs to each access node. One input arc (called the *enable* arc) determines the object, performer, or initiator to be enabled (see section 5.1). Tuples flowing on this arc are assumed to be events with one domain containing the name of an entity (object or process). A *start event* specifies the entity to be enabled; a subsequent *stop event* specifies that the entity is to be disabled. Note that there is a potentially long delay before the tuple actually arrives at the access node, since the tuple was the result of an arbitrarily long sequence of calculations. Hence enabling an event based on the occurrence of a different event cannot be guaranteed to be timely in all cases.

If the access node is associated with a sampled event, a second input arc (called the *trigger arc*) determines when sampling is to occur (trigger arcs are not necessary for access nodes associated with traced events). Whenever a tuple arrives on the trigger arc, a command is sent from the access node to the resident monitor to perform the sampling. The one domain

of the tuple specifies which object is to be sampled. Again, there is the proviso that a delayed trigger tuple will cause the sampling to also be delayed.

The two types of input arcs (enable and trigger) can be merged into one by specifying the semantics so that a start event enables the event on the object (for a traced access node), a stop event disables the event on the object (also for a traced access node), and either causes a sample to be taken for a sampled access node. In this characterization, the one input arc is called the *control* arc.

The presence of the control arc on the access nodes allows the monitor to construct subnetworks to compute the entities to be enabled and the samples to be taken. By allowing these functions to be performed in the same manner as the calculation of derived relations, the mechanisms already present for manipulating these networks can also be applied to these areas. In particular, the optimization strategies discussed in section 7.2 are completely applicable to the subnetworks controlling the enabling and sampling activities of the access nodes they are connected to.

6.4. Operator Nodes

Operator nodes perform the computation on the event records flowing from the resident monitor. Some nodes compute new domains in the output tuple from existing domains in the input tuple; the rest map the input domains to the output domains in some fashion. The algorithms in the operator nodes, while performing standard relational operations such as join and projection, are nevertheless quite different from their static database counterparts, since they have been tuned for the incremental update of temporal relations.

There are about a dozen generic operator nodes, varying greatly in complexity. Appendix C describes each in more detail.

6.5. Node Interconnection

In order to process the tuples, the access and operator nodes must be instantiated from the available generic nodes and combined into an update network with the arcs corresponding to paths over which the individual tuples flow. Three primitive constructor functions are provided. The *create* operation

    (create genericnode name (param$_1$ ... param$_n$))

creates an instantiation of the genericnode with the specified instantiation parameters and associates the name name with it. The *link* operation

```
(link fromnode tonode side)
```

creates an arc from fromnode to the side of tonode. The side is either left, right, or control

(see section 6.3). The update network must be a directed acyclic graph (dag); the link operation ensures that no cycles are created. The third operation

```
(unlink fromnode tonode)
```

simply removes an arc from the update network. Nodes are garbage collected by the unlink operation when the last arc is removed.

These operations, along with the predefined collection of generic nodes, define the update network abstraction as seen by the query translator. Specifically, the code generated by the translator is a sequence of create, link, and unlink operations. There are two remaining issues pertaining to the update network: how is the network abstraction supported (considered in section 8.6), and how is the query translated into an update network (dealt with in the next chapter). At this point, the problem given in chapter 3:

> **Problem:** How can the dynamic incremental updating of temporal relations be implemented effectively, and in such a manner that the facilities supplied by the data collection mechanism are also used effectively?

is reduced to a result and a smaller problem:

> **Result:** An update network can be used to implement dynamic incremental updating of derived relations. The network is composed of access nodes, which interface effectively with the data collection mechanism, and operator nodes, which carry out the desired computations.
>
> **Problem:** How can the update network be implemented to efficiently process the incoming event records?

The next problem to be faced is now more specific:

> **Problem:** How can the knowledge contained in the monitor itself and in the incoming event records be used to direct the translation of user queries into efficient, correct update networks?

Chapter 7
Generating the Update Network

The remote monitor has two primary functions - generating an update network from the user's query, and executing the update network to process event records flowing in from the resident monitor. The update network must be correct and it must be efficient. Update network generation is therefore a vital aspect of monitoring and, to be effective, should use all the knowledge present in the monitor. The network generation component has two sources of knowledge: the information contained in previously processed event records, and the knowledge embedded in the monitor's algorithms. The purpose of this chapter is to describe how these two sources of knowledge can be applied to the task of generating the update network.

7.1. Generating an Initial Network

The query is parsed by a LALR(1) parser generated from the TQuel grammar by the yacc parser-generator [Johnson 75], building a standard parse tree. Semantic analysis consists of resolving the names for tuple variables, relations and domains, type-checking the expressions, and inserting semantic information into the tree. The parse tree is then converted to a relational operator tree (or, equivalently, an initial update network, see section 6.1) by the following process, illustrated using the following query from section 3.7:

```
retrieve Catch
where A.IterNum = B.IterNum
when A.start; B.start
at B.start
```

1. Collect all the referenced tuple variables.

    Two tuple variables appear in this query: A and B.

2. Build a tree of Cartesian Product operator nodes of all the relations associated with the referenced tuple variables.

    The tree is quite simple: A × B, requiring one Cartesian Product operator node.

3. Create ApplyOp operator nodes for all expressions in the parse tree, except those in the where and when clauses.

```
There are no such expressions in this query.
```

4. Create a series of Selection operator nodes, one for each disjunction in the where clause, after converting the where clause to a conjunction of disjunctions (*normal form*[14].)

```
     The where clause is already in normal form, requiring one Selection operator
node.
```

5. Create one or more Selection operator nodes for the when clause, after converting to normal form.

```
     The when clause is already in normal form, requiring one Selection operator
node.
```

6. Create a Projection operator node from the target list of the parse tree, including the at, start, and stop clauses.

```
     The projection node will extract only one domain, Catch.start.
```

The resulting initial update network is shown in Figure 7-1. The update network produced by this process is not quite complete. The rest of this section will discuss the modifications required for semantic correctness.

7.1.1. Universal Relations

As discussed in section 6.3, the control arc is included on access nodes to determine which objects are to be sampled or traced. The monitor must attach a source of objects to the control arc of each access node in the network; otherwise, no events would be enabled, no samples would be taken, and no event records would be generated. For this purpose, a *universal relation* is defined for each object type. A universal relation contains exactly one domain denoting the set of objects of a given type known to the monitor. Whenever the monitor is informed of a new object, the object is added to the appropriate universal relation. The control input of each access node is connected to a universal relation determined by the type of the access node. An access node associated with a sensor in the file system would have its control input connected to the universal relation of file system processes. If the sensor was instead associated with a file object, the control input would be connected to the universal relation of files.

Connecting a universal relation to the control arc of each access node implies that, for each primitive relation referenced in the query, all relevant objects known to the monitor will have the associated sensor enabled for the object. Sometimes this is desired; for example,

---

[14] The multiple Selection operator nodes provide flexibility in the optimization phases described later.

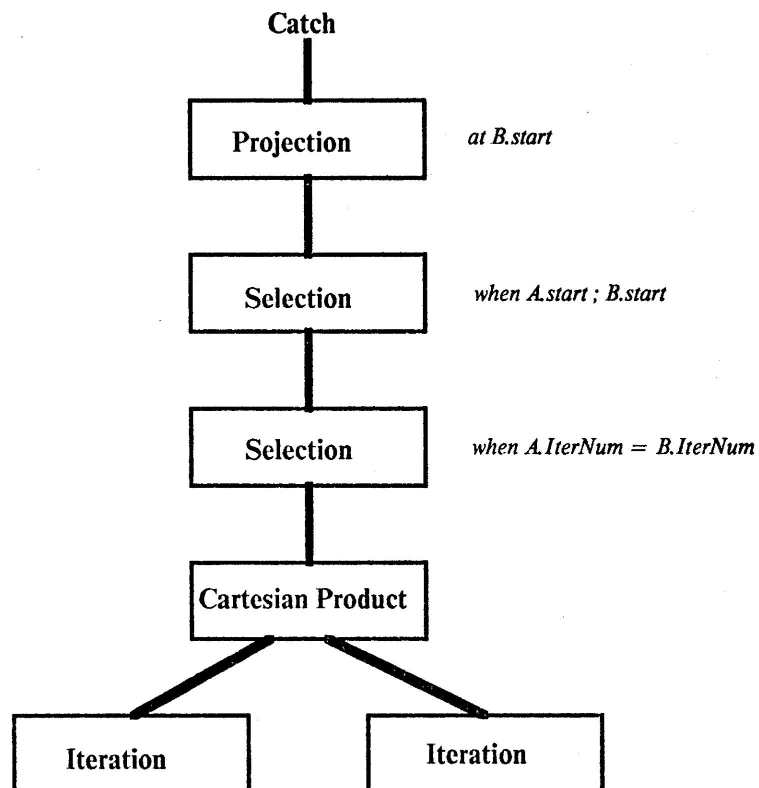

Figure 7-1: An Initial Update Network

determining the system utilization requires context switch events to be enabled on all the processes in the system. In general, however, the monitor must be much more selective in which sensors are enabled, with the selection criterion extracted from the query itself. This is done by first attaching the universal relation to each control input, assuming initially that all sensors are to be enabled. Then, the update network corresponding to the query is converted into a more efficient one using a collection of transformations discussed later. The new update network will probably have an entire subnetwork replacing the universal relation connected to the control input. In particular, selection nodes are migrated down toward the access nodes, often coming to rest below the access node in the subnetwork connected to the control input. Examples of this optimization will be given later in this chapter.

7.1.2. Compensation

Compensation is necessary because the act of monitoring usually perturbs the system under study. For instance, the monitoring of processor utilization may decrease the utilization, since some of the processing involves monitoring rather than running user jobs. If the monitor has some measure of the perturbation the monitoring of each event has on the other events being monitored, it can compensate by correcting for this perturbation.

Although it is impossible to compensate totally for the operation of the monitor, it is sometimes feasible to perform a few gross corrections. For instance, by knowing how much processor time went into generating event records, the monitor can adjust the computed processor utilization in a straightforward manner. The monitor can determine the approximate overhead of event record generation using the potentially large amount of information it has on its actions:

- which sensors executed when (from the event records themselves);

- how much data was collected (also from the event records);

- the internals of the sensors (from the descriptions of the sensors); and

- the effect each sensor has on other sensors (presumably also from descriptions of the sensors).

The monitor, when generating the update network from the user's query, may add special compensation operator nodes to the network. The information from the sensor descriptions is available at network generation time; the information from the event records must be derived dynamically from the tuples as they flowed into the compensation node. Thus, one input to the compensation node would be the tuples to be compensated (the *primary* tuples); the other input(s) would be for the (*secondary*) tuples, whose collection perturbed the values of the primary tuples.

As just described, compensation nodes would not work, for there would always be the possibility (actually, the probability) that the primary tuple would have come and gone before the secondary tuples arrived at the compensation node. There are at least two solutions to this problem. In one approach, the network itself would ensure that the tuples arrived in the correct order. It is unclear how the network would enforce the correct ordering. The other approach is at the other end of a local-global spectrum: instead of adding complexity to the network, the nodes requiring a specific ordering of the incoming tuples would achieve that ordering internally. In this approach, the compensation node would store the primary tuples until the correct secondary tuples had arrived, and then output the corrected primary tuples. The latter approach was adopted here, for reasons to be given later.

Unfortunately, this approach introduces another problem. The remote monitor receives tuples periodically as they are retrieved from internal buffers (i.e., the receptacles) within the system being monitored (see section 5.2 and appendix D). The tuples are then sent to the appropriate access nodes for insertion into the update network. Due to the presence of multiple buffers and the unspecified manner in which these buffers are emptied, the incoming tuples are arbitrarily ordered in time. In particular, the differences in time between subsequent tuples can be arbitrarily large, since some buffers fill up very quickly (say, every second) and others quite slowly (say, every hour). This situation implies that the compensation node must at any point be prepared for a secondary tuple arriving much later than a primary tuple. Primary tuples must be stored indefinitely in preparation for such a possibility.

There are two possible solutions to this dilemma. One solution is to have the resident monitor remove event records in an orderly fashion so that the buffers are emptied regularly and the tuples can be ordered. The other solution relies on the generation of *checkpoint* tuples to aid in the temporal ordering of tuples. Since the latter solution does not put any restrictions on the gathering of tuples by the resident monitor, and allows the temporal order to be changed at will in the update network, it was adopted in the implementation.

7.1.3. Checkpoint Tuples

A checkpoint tuple has the semantics that all tuples of a particular class following the checkpoint tuple will have a time value after that of the checkpoint tuple. A class might contain those tuples of a particular event type, or those tuples pertaining to a particular object or sensor process, or those tuples generated on a particular processor, or even all the tuples generated by the system, depending on how the buffering and movement of tuples was implemented in the resident monitor[15]. Most operator nodes simply echo checkpoint tuples on their outputs. Binary operator nodes must generate output checkpoint tuples as a function of checkpoint tuples from the two inputs. To preserve the semantics of the checkpoint, each binary operator node behaves in the following manner. First, the node accepts and stores tuples from both inputs until a checkpoint has been received from both. Assume that the left input had a checkpoint at t_1 and the right at t_2, with $t_1 < t_2$. Also assume that the last checkpoint tuple output by this node had a time t_0. The node processes all the internally stored tuples in $[t_0, t_1)$, then outputs a checkpoint tuple with a time t_1. All tuples from both inputs with times less than t_1 are purged from internal storage, and the processing continues by again accepting input tuples.

It is useful to examine how checkpoints relate traditional and temporal databases. A traditional static relation, which models reality for an instant or period of time (after which it is updated to model the revised reality), is simply the subset of the tuples of a temporal relation

---

[15] In both resident monitors implemented to date, a checkpoint applies to the entire system.

valid during that instant or period of time. Checkpoints break the stream of tuples into consecutive intervals, with tuples in a particular interval all generated after one checkpoint and before the next. The Cartesian product operator performs a series of somewhat conventional Cartesian products over these intervals. Tuples in one interval from the left input will be concatenated with overlapping tuples in only a few intervals from the right input[16]. The overall effect is to divide the relation temporally into intervals, and apply the operations to one interval at a time.

One issue still remains: when are checkpoint tuples generated? Internal storage requirements in the resident monitor dictate that checkpoints be generated regularly in short time intervals. Also, because tuples are potentially delayed until the next checkpoint, responsiveness to events enabled by tuples flowing into a control input of an access node from lower in the network demands that checkpoints come often. However, since checkpoint tuples themselves require processing resources, they should not be generated too often. This tradeoff is complicated when checkpoints are associated with classes of tuples, instead of having each checkpoint refer to the entire system. The optimal frequency of checkpoints is in general a function of the storage capacity of the resident monitor, the tuple processing capability of the update network, the responsiveness desired by the user, and the previous tuples generated by the operating system and user programs. It is desirable to have the update network have control of this aspect, as it does with event enabling and compensation. One way to control the checkpoint frequency is to provide a unique checkpoint access node. Whenever a tuple arrives at the control input (see section 6.3) of this access node, a command would be sent to the resident monitor to generate a checkpoint. By connecting a clock access node to the checkpoint node, checkpoints would be generated regularly. More generally, this arrangement allows checkpoints to be generated as a result of arbitrary processing of event records. A checkpoint tuple would then be similar to a sampled event, with rather unique semantics for the update network. The approach taken in the current implementation is to instead simply generate checkpoints regularly, every few seconds, as specified by the user.

7.1.4. Other Details

The semantic analysis must also concern itself with several issues which, although straightforward, nevertheless should be mentioned:

- The aggregate operator node (see appendix C) assumes the presence of domains indicating (a) which partition the tuple is a member of, (b) the value of the expression the aggregate is to be applied to, and (c) a Boolean indicating whether this tuple is to be included in the aggregate, corresponding to the by clause, expres-

[16] This is actually a simplification; see section 7.2.3 for the full story.

sion, and where clause, respectively, in the aggregate clause (see section 3.6). These domains may already exist in the tuple, or they may need to be computed by ApplyOp operator nodes lower in the network. Appendix C, in its discussion of the AggrOp operator node, includes an example illustrating the use of an ApplyOp operator node to accommodate a where clause.

- If a start or stop clause is omitted, the default (see section 4.5.2) must be inserted into the parse tree before the update network is generated.

- The operator nodes expect domain indices as arguments, rather than domain names. Thus, the semantic analysis must keep track of the number of domains and the content of each domain of the tuples flowing over each arc of the network. This accounting is complicated slightly by those operator nodes that compute new domains. For efficiency, lifetime analysis may be used to indicate which domains may be reused, thereby reducing the tuple size.

- Events associated with a traced primitive relation are automatically converted to periods by inserting an EventToPeriod operator node above the access node (see appendix C).

Although the update network produced by this process is semantically correct, it is probably too inefficient to execute directly. The rest of this chapter will discuss ways to improve both the time and space efficiency of the update network while maintaining its correctness.

7.2. Efficiency

In order for the update network to operate in an efficient and timely manner, the arrival of an input tuple at each node should cause a small, relatively uniform amount of processing. If significant processing is required for each tuple, the monitor will be severely constrained as to the maximum incoming tuple rate it can handle. Widely varying processing times cause "jerky" behavior by the network, which may be acceptable when storing the resulting tuples for later analysis, but are clearly unacceptable when the tuples are being viewed in real-time by the user.

Minimizing the internal storage of each node is also important. The ideal, no internal storage at all, is possible only for a few nodes such as ApplyOp and Selection. On the other end of the spectrum, some nodes, such as the Cartesian product, require unbounded internal storage. Clearly this is an unacceptable situation.

Four related approaches are available to increase the efficiency of an update network. One

approach applies graph transformations to the update network, mapping an inefficient network into a more efficient one. The second approach exploits the presence of the implicit time domain by temporally ordering the tuples flowing across the arcs in the network. The third approach places restrictions on the tuples the Cartesian product can produce, by limiting the generality of the <when clause>. The last approach is concerned with the scheduling of ready nodes (those with tuples on their input arcs). These approaches, applied in concert, offer the possibility of increasing the time and space efficiency of an initial update network by several orders of magnitude. A fifth approach, optimizing the update network *after* it has been generated, will be discussed in section 8.6.2.

The Cartesian product provides a convenient example of all four approaches. A conventional Cartesian product relational operator concatenates a copy of every tuple from one relation with a copy of every tuple from a second relation. If there are n_1 tuples in R_1 and n_2 tuples in R_2, there there will be n_1n_2 tuples in $R_1 \times R_2$. However, when the tuples represent periods of time, they should be concatenated only if they overlap in time, i.e., only when their start times strictly precede their stop times (the when clause allows other temporal criteria to be used; overlap is the default and most commonly used criterion.) A tuple from the first relation will in general overlap only a small fraction of the tuples from the second relation.

The space and time efficiency of the Cartesian product is strongly dependent on the number of tuples that must be processed and stored internally, which in turn depend on four factors:

- the number of tuples in the underlying relations;
- the temporal ordering of the two input tuple streams;
- the number of tuples from one input which are concatenated with a tuple from the other input; and
- the relative synchrony of the input streams.

The first factor depends solely on the computation process generating the events; the second factor is under the control of the update network; the third factor is determined by the exact semantics of the sequence operator in the when clause (to be discussed in more detail below); and the fourth factor depends in part on the scheduling of ready nodes. Graph transformations can be used to reduce the number of tuples participating in the Cartesian product. Through the use of conversion operator nodes, the update network can alter the temporal order of the tuple stream. A modified Cartesian operator node can greatly reduce the number of output tuples generated. And finally, the scheduling of ready nodes strongly affects the relative synchrony of the tuple streams. Each of these approaches will now be discussed in more detail.

7.2.1. Graph Transformation

The first method, graph transformation, is used extensively in conventional relational database systems [Ullman 82], and is a primary reason why the relational approach is now a viable one. Graph transformations map an algebraic operator tree (i.e., the update network used here) into a more efficient tree. Each transformation consists of a predicate and a replacement. The predicate specifies the properties a node or tree of nodes must have in order to satisfy the predicate. The replacement is a node or subtree which replaces the subtree matching the predicate. The transformations are repeatedly applied in a specific order to the operator tree until no predicate can be satisfied. The collection of graph transformations can be viewed as a *production system* [Forgy 79].

One example is a transformation that maps the subtree $\sigma_F(R_1 \times R_2)$, i.e., the Cartesian product of R_1 and R_2 followed by a selection by the Boolean function F, into $\sigma_{F_1}(R_1) \times \sigma_{F_2}(R_2)$, if F is of the form $(F_1 \wedge F_2)$, and F_i only involves domains in R_i. Such a transformation preserves the semantics of the expression, yet can increase the efficiency tremendously. The Cartesian product has a time complexity of $O(n_1 n_2)$, where n_i is the number of tuples in the ith input relation. In this example, the Cartesian product is provided with fewer tuples from the underlying relations, with the magnitude of the reduction depending on the selectivity of F_1 and F_2. If F_1 and F_2 each eliminate 90% of their input tuples (a common occurrence), then the Cartesian product in the transformed tree will produce approximately 1% of the tuples produced by the Cartesian product in the original tree. This example illustrates the use of one transformation to reduce the execution time by two orders of magnitude and the internal storage for the nodes in the network by one order of magnitude.

For the most part, the transformations used for operator trees in conventional data base systems, such as the one just described, can be applied directly to update networks. In addition, transformations developed for optimizing compilers, such as common subexpression elimination, are relevant. A few additional transformations not having relational operator analogues have been developed for the specialized nodes of update networks, particularly the access nodes. These transformations will now be examined, followed by an integrated example.

The first transformation makes use of the control input of access nodes (see Figure 7-2a, primitive relations are represented by dark rectangles).

This transformation may be applied if

1. No domains other than domain *k* are used from B;

2. A is a primitive relation; and

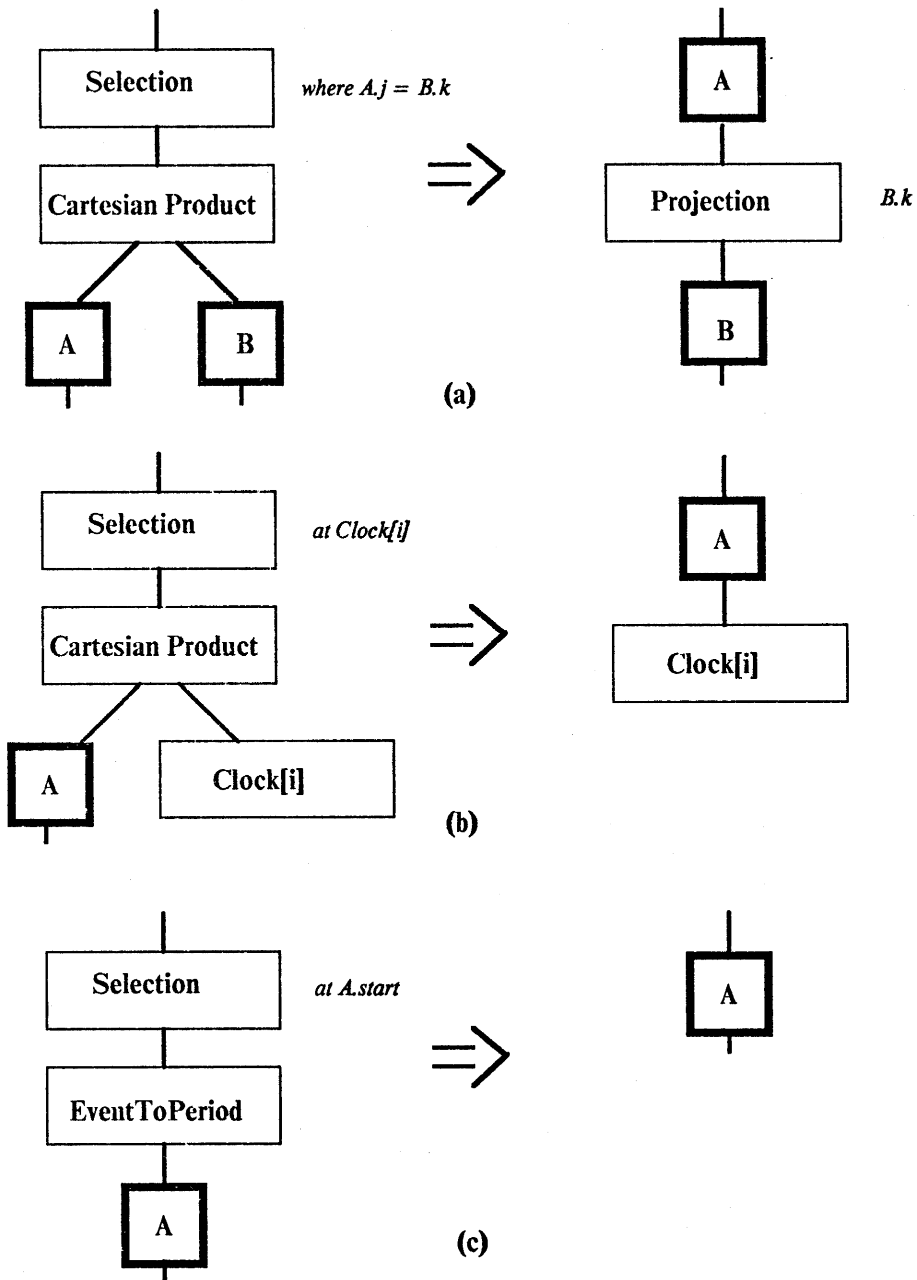

Figure 7-2: Monitor-Specific Transformations

3. Domain A.j is A.Process, if A is associated with a sensor-traced event, or

A.Object, if A is associated with an object-traced event (similar restrictions apply if A is associated with a sampled event).

If these criteria apply to both A and B, the one outputting the most tuples should be moved to the top.

The idea behind this transformation is that domain *k* in B is being used to select tuples out of the primitive relation A. Instead of generating all of A and then discarding most of it, as the original update network does, the modified network uses B.k to generate only the correct tuples in the first place.

The transformed update network has many advantages over the original network. The selection, universal, and costly Cartesian product operator nodes are eliminated (the projection nodes in the transformed network may also be eliminated if the k domain is the first one in the relation). By using the control input to the access node for A, only those objects or sensors found in B are enabled, thereby significantly reducing the number of event records generated.

The second transformation is simply a variation on this theme (Figure 7-2b). The only restriction is that A be associated with a sampled primitive relation. This transformation recognizes that tuples of a relation are desired only at clock ticks. The third transformation (Figure 7-2c) recognizes that only the start event of a traced primitive relation is necessary, so the events are never converted to periods.

As an integrated example, consider the follow query,which uses the primitive relation EXECUTIONSTATUS (Process, State) and the derived relation F (Process):

```
retrieve StopRunning (E.Process)
where E.State = Done and E.Process = F.Process
at E.Start
```

This query determines the time when each process in F stopped.

The initial update network is shown in Figure 7-3a.

The first transformation eliminates a selection, a Cartesian product, and a universal node (Figure 7-3b); no additional projection is needed since F contains only one explicit domain. The second transformation eliminates an event to period node (Figure 7-3c). The transformed network contains half as many nodes as the original network; the nodes are all memoryless; and the tuples are all events. Only the processes in F will have the ExecutionStatus sensor enabled. Thus, the transformed update network is much more efficient than the original one. A more involved example, complete with performance measurements, appears in section 8.6.

Transformations provide one way to encode knowledge concerning monitoring-specific

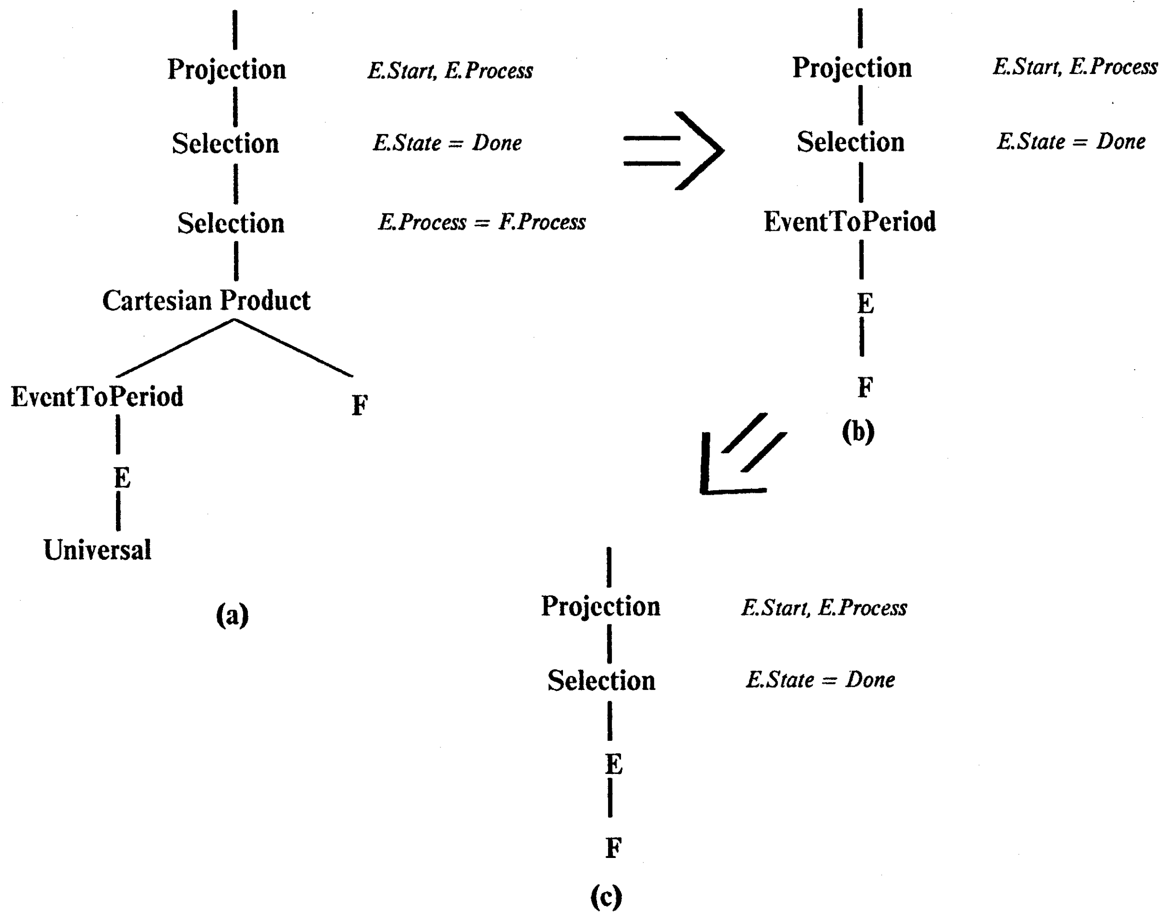

Figure 7-3: Applying Multiple Transformations

aspects of the update network. The three transformations presented above concern such mechanisms as the control input to access nodes, sampled primitive relations, and the support of primitive period relations associated with traced sensors. As new mechanisms are added both to the update network and the resident monitor, new transformations may be developed so that the network generation component can utilize these additional mechanisms.

7.2.2. Temporal Order

There are four ways to order periods temporally (event tuples can of course be ordered temporally only one way):

1. (non-overlapping) period tuples ordered by their start and stop times;

2. start-period and stop-period event tuples;

3. period tuples ordered by their start times; or

4. period tuples ordered by their stop times.

Different algorithms require different temporal orders. The operator nodes can be divided into two classes with regard to temporal ordering: those that are *memoryless*, i.e., do not require internal storage, and those that do require internal storage. Space and time complexity are correlated at this level of detail--the greater the internal storage, the more processing is required to search or augment the internal storage when a new input tuple arrives. Of course, once the correct temporal ordering of input tuples for a particular operator node has been determined, then space can be traded off for time. Unary operator nodes whose processing only involves the current input tuple, (e.g., ApplyOp, Selection) are memoryless, so their storage requirements do not depend on temporal order. Their execution efficiency is also invariant with respect to temporal order.

The rest of the operator nodes all require internal storage, and therefore are sensitive to temporal order. Some operator nodes require a specific temporal order; the conversion may be done either with special operator nodes, or as a side effect. Representation (1) occurs naturally when traced events are converted to periods, and is a special case of representation (2); (1) can be converted to (2) trivially by converting each incoming period tuple into consecutive start-period and stop-period tuples. If the periods do not overlap, the same holds true for representations (2) and (3). To convert from (3) to (2), maintain a list of tuples ordered internally by stop time. Whenever a new tuple comes in, first output stop-period tuples for all internally stored tuples whose stop time precedes the start time of the input tuple, then output a start-period tuple for the input tuple. Finally, store the input tuple in the correct position in internal storage. A similar algorithm maps (2) into (3). The internal storage contains all the tuples valid at the "current" time (the start time of the current input tuple); some operator nodes require this information anyway in their calculation. Representation (4) is unacceptable, since a node having an input tuple stream in this representation must at any point be prepared for an input tuple arriving much later yet which started before the current time. Tuples must be stored indefinitely in preparation for such a possibility.

The central point in this discussion is that, for each operator node, some temporal orders

are definitely unacceptable, some are acceptable, yet imply additional processing, and one temporal order is optimal. The goal is, given a particular update network, determine a temporal ordering for each arc, consistent with the operator nodes connected to that arc, which minimizes execution time and internal storage for the entire network. This task is made more flexible by providing several versions of each operator node which take as inputs and produce as outputs different temporal orders. Additional flexibility (and complexity) is introduced by operator nodes which change the temporal order. The operation of the ordering operator nodes is to store the incoming tuples until a checkpoint tuple arrives, then output the stored tuples in the appropriate order, using the conversion algorithms given previously, followed by the checkpoint tuple. These operator nodes, although themselves expending time in execution, could potentially increase the efficiency of the entire network if inserted at the correct positions (a similar case could be made for eliminating duplicate tuples).

Determining the optimal ordering for a query is in general impossible, since an analysis of time and space efficiency for the query requires knowing the number and content of tuples which will be flowing over each arc. However, it is sometimes possible to determine an ordering for each arc which is optimal for widely varying tuple counts. There also exist networks where a particular ordering for each arc is optimal for any set of tuples flowing through the network. One heuristic which works well is to require the access nodes to order their output tuples by start and stop times, if the periods do not overlap, and otherwise by start times. Two versions of operator nodes are required, differentiated by the tuple orderings they expect. Determining the temporal ordering at internal arcs is now straight-forward, since the ordering of the output of an operator node is fixed once the ordering of the input arcs has been determined. This topic clearly requires more study; the above analysis constitutes a first cut at the problem.

7.2.3. Limiting the Semantics of the When Clause

The innocuous sequence operator in the <when clause> of the TQuel retrieve statement greatly influences the efficiency of the corresponding update network. The problem occurs in the Cartesian product operator node. Recall that the Cartesian product concatenates all tuples from the left input with all the tuples from the right input. Since all the tuples that have arrived at any point in time from one input are to be concatenated with future tuples from the other input, no tuple can ever be removed from internal storage. This is clearly an expensive option, both in terms of execution time and storage requirements.

However, there is an optimization which makes the Cartesian product feasible. The <where clause> and <when clause> serve to eliminate some or most of the tuples produced by the Cartesian product. If the <when clause> does not contain a sequence operator, then all tuples where the underlying tuples did not overlap will be eliminated. Hence, in this case, the Cartesian product operator node need only store incoming tuples internally as long as there is

a possibility of overlap with tuples from the other input. Once it is clear the tuple will not contribute to any more output tuples, it can be eliminated from internal storage. The node can utilize checkpoint tuples or the temporal order of the incoming tuples to this end. Since the incoming tuples will be stored internally for only a short amount of time, rather than indefinitely, the space and time requirements will be greatly reduced.

Unfortunately, not allowing the sequence operator is a rather severe restriction. In the following discussion, a series of five implementations of the Cartesian product will be examined, each with a different semantics for the sequence operator in the when clause, trading generality for efficiency. The implementations differ in the tuples that are considered to be consecutive, and thus in the number of tuples that must be retained for possible concatenation with tuples arriving later.

The most restrictive, and thus the most efficient, implementation (1) of the Cartesian product outputs concatenations of periods that overlap. This is the implementation discussed above, where the semantics of the sequence operator is undefined, because it is not allowed[17].

The next implementation (2) allows the sequence operator in the when clause, but requires at least one period to be in effect at all times. Tuples are stored internally as long as a partial overlap is possible.

Implementation (3) does not require overlap, but only concatenates tuples from one input with tuples from the other input which overlap *or* which are closest temporally (*closest-neighbor* tuples). Hence, for a given input tuple, if no tuples from the other input overlap the tuple, then only two output tuples will be generated: the original tuple concatenated with the tuple from the other input occurring directly *before* the original tuple, and the original tuple concatenated with the tuple occurring directly *after* the original tuple.

Implementation (4) uses the closest neighbor approach only for duplicate tuples[18]. The Cartesian product must keep one copy of each tuple around (potentially the entire database).

The last implementation (5) supports the sequence operator in its full generality. It is even less efficient than (4).

Since implementations (1) and (2) seem overly restrictive, and implementations (4) and (5) are overly inefficient, implementation (3) (closest-neighbor) offers the best compromise be-

---

[17] Although the sequence operator is not explicitly allowed, it may be used by the monitor to implement the parallel operator (see section 4.2.2).

[18] Note that duplicate tuples as used here match only on the explicit domains; they still occur at different times.

tween generality and efficiency. Of course, implementations (1) or (2) should be used in the update network for a query not utilizing the full generality of the when clause.

7.2.4. Node Scheduling

At any point in time, there are potentially many nodes with tuples on their input arcs, ready to be invoked. This situation raises the issue of *scheduling* the nodes: determining the order in which they are invoked[19]. There are two general scheduling disciplines: depth-first and breadth-first. In depth-first scheduling, as each node produces a tuple on its output arc, the tuple is immediately sent to the nodes the original node was connected to. In breadth-first scheduling, the output tuples are queued, and sent to their destination nodes in the order of their creation. Breadth-first scheduling ensures that the incoming tuple streams stay at approximately the same level as they travel through the network. This may result in a more favorable synchrony of the tuple streams, thereby reducing the internal storage requirements of the network. However, such a reduction has a cost: the tuples on the arcs must be stored until it is their turn to be sent to the appropriate nodes. Depth-first scheduling uses only the internal storage of the nodes in the network; the arcs have no storage capacity.

Both scheduling disciplines require mechanisms for handling checkpoint tuples, for converting between temporal orders, and for internal storage of tuples at nodes which are not memoryless. The central issue becomes the comparison of the increased processing resulting from managing queues for breadth-first scheduling versus a larger number of tuples in internal storage for depth-first scheduling. This issue is discussed further in section 8.6.2.

7.2.5. Other Issues

Several of the operator nodes may generate duplicate tuples[20]. A good example is the Projection operator node. Even if all the incoming tuples are unique, the output tuples may not be unique, since one or more of the domains have been discarded by the operator node. Duplicate tuples matter semantically only when they are displayed or when they participate in certain aggregate functions such as Count. The former is usually dealt with by allowing the user to specify whether or not duplicates are to be displayed (c.f., Quel [Held *et al.* 75] and Sequel [Astrahan&Chamberlin 75]). The participation of duplicates in aggregate functions was discussed in section 4.4.1. However, when considering performance, duplicates are of greater concern. The usual approach, adopted here, is to eliminate duplicates whenever

---

[19] We are assuming that the update network executes sequentially. One appealing alternative is the parallel execution of the update network.

[20] Here, the term duplicate is used to mean identical in all domains, including the implicit time domain.

easily done in the course of other processing (such as sorting), and to leave them otherwise. Any novel approaches for handling duplicates in standard relational databases would probably also apply to temporal databases.

A second issue relating to efficiency is the coupling between the remote and resident monitors. It is fair to state that the transmission time between the resident and remote monitors will be one or more orders of magnitude slower than the time required to move a tuple through the update network[21]. The transmission time becomes crucial if the processing of a tuple results in a sample being requested by the remote monitor, or an event being enabled or disabled. In these cases, the monitor might be able to move the critical portion of the update network over to the resident monitor.

As one example, suppose that the system utilization was requested. This query would result in an update network with the universe relation for processes (containing all the processes the remote monitor was aware of) connected to the control input of the ContextSwap access node, specifying that context swaps for all processes were to be monitored. The same effect could be obtained by having the resident monitor instruct the process creator to automatically enable the ContextSwap for each process it constructed. In effect, a portion of the update network is being performed in the system being monitored, rather than in the remote monitor. This optimization could be implemented using one or more graph transformations. An open question is, what mechanisms, such as automatic event enabling, are useful for implementing specialized portions of update networks in the resident monitor?

There is another aspect regarding efficiency which concerns the delay between the occurrence of an event and the instant to monitor makes the user aware of the event. Most monitoring systems are *off-line* monitors, in that the data is analyzed after the process has completed executing. Some monitors are *on-line*, since the data is analyzed concurrently with program execution. The goal is a *real time* monitor, guaranteeing a rather short delay between the occurrence and the display of the event. The monitor described here is an online monitor, due to the presence of operator nodes which are not memoryless. Such nodes may delay typles for arbitrary lengths of time, invalidating any response time guarantees.

---

[21] In the implementation described in chapter 8, the times differed by a factor of several hundred.

7.3. Summary

This chapter has considered the task of applying as much knowledge as possible to the generation of update networks from user queries. The process is composed of three stages:

1. Generating an initial update network via the relational operator tree for the query;

2. Modifying this network so that it correctly computes the desired information; and

3. Modifying the network so that it executes efficiently.

The knowledge used by the monitor in this process includes:

1. information on the number and types of domains for all relations;

2. additional information on the sensors, such as whether they are traced or sampled;

3. information on the available universal relations;

4. how the sensors affect data collected by other sensors;

5. how to incorporate checkpoints into the update network;

6. defaults for the where, when, start, stop, and at clauses;

7. the collection of graph transformations;

8. information on the temporal orders supported by each operator node;

9. heuristics for determining the temporal order for each arc in the network;

10. other attributes of operator nodes, such as whether they are memoryless; and

11. varieties of the Cartesian product operator node, and how to select the correct variety.

Such knowledge is available from several sources, primarily the algorithms embedded in the monitor, the incoming event records, and the data structures constructed from these event records. At this point, an answer is now available to the problem stated in the previous chapter:

Problem: How can the knowledge contained in the monitor itself and in the incoming event records be used to direct the translation of user queries into correct, efficient update networks?

Result: The primary techniques available for generating update networks are (a) the translation of the query into a relational operator tree and then into an initial update network, (b) the translation of this network into a correct network, and (c) the use of graph transformations, temporal order, and node scheduling to greatly increase the efficiency of the network. The system-dependent aspects can be embodied in the primitive relations and associated access nodes.

The development of such techniques leads to the final major problem to be addressed in this dissertation:

Problem: What is involved in implementing these proposed solutions? Can the relational view be supported in concrete terms by a functional monitor?

Chapter 8
An Implementation

The previous chapters have focussed on the support of the relational model in monitoring distributed systems. This chapter describes in some detail one realization of the solutions and techniques proposed in those chapters. Many past efforts in monitoring were based on models that lent themselves to efficient implementations; this thesis attempts to demonstrate that one can utilize conceptualizations convenient to the user (i.e., the relational viewpoint) without paying for it in terms of a hopelessly slow or complex monitor. A viable implementation is necessary to support this assertion.

A minimal monitor has been implemented. Although certain components have been omitted, and no component has been completely implemented, all aspects have been carried far enough to demonstrate feasibility. The remainder of this chapter will describe in some detail the design and performance of this implementation. Appendix E demonstrates the use of the monitor by stepping through the complete task of monitoring an application program, the one described in section 3.7, from specifying the sensors to viewing the derived relations.

8.1. General Structure of the Monitor

Previous discussions have characterized the monitor as consisting of two components, a *remote monitor* and a *resident monitor*. The remote monitor is system independent, accepting user queries, processing the incoming event records, and displaying the derived information, whereas the resident monitor generates, collects, and sends the event records to the remote monitor. The remote and resident monitors each contain subcomponents: the remote monitor consists of the relational database system, the monitor core, and the display module; the resident monitor has a system dependent structure (see Figure 8-1).

In the implementation, the remote monitor runs on a Vax under Berkeley Unix [Ritchie&Thompson 74]. The system being monitored is Cm* [Fuller *et al.* 78, Swan *et al.* 77], a tightly-coupled multiprocessor composed of 50 DEC LSI-11's and a substantial amount of memory. All memory in the system is potentially accessible to all processors through five microprogrammed controllers called *KMaps.* The microcode in the KMaps supports arbitrary memory addressing mechanisms and operating system primitives. The fortunate presence of

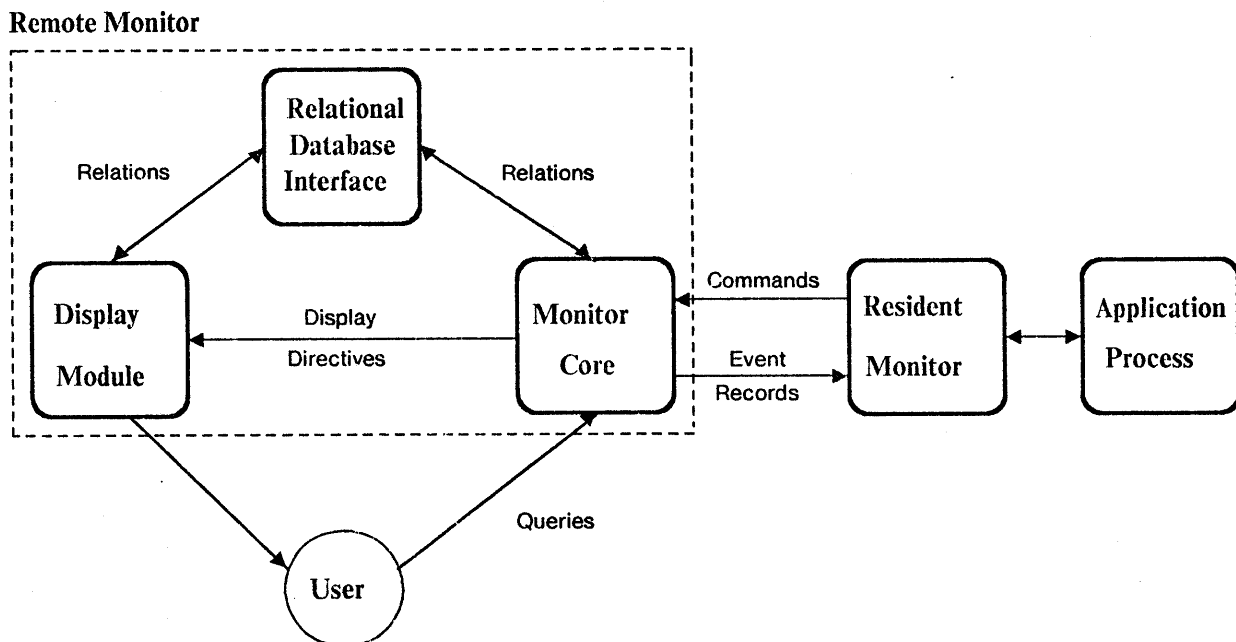

Figure 8-1: Components of a Distributed Monitor

two recently developed operating systems for Cm* provides an excellent opportunity to study the interactions of the remote monitor with different resident monitors. Two resident monitors were implemented, one on StarOS [Jones *et al.* 78, Jones *et al.* 79, Gehringer&Chansler 82] called *StarMon*, and one on Medusa [Ousterhout *et al.* 80] called *Medic*. Both operating systems are message-based, object-oriented, and fully distributed. Both provide a stable environment of many asynchronous, concurrently executing processes, communicating through messages or shared memory. This environment allows the monitor's ability to interface both with distributed systems (using applications which communicate only through messages), and with traditional multiprocessors (using applications which also communicate through shared memory), to be validated.

The remote monitor on the Vax communicates with the resident monitor on Cm* over an Ethernet [Metcalfe&Boggs 75], a high bandwidth (3 MBaud) network. Since the Ethernet is connected to most of the available machines, the remote monitor may be used to monitor applications in different machines. Interfacing with another system requires the creation of a new resident monitor, as well as the definition of the primitive relations of the new system. In particular, applications running on different machines, using radically different operating systems, can be monitored once the appropriate resident monitors have been constructed.

The remote monitor was partitioned into modules by considering the required functions. The *display module* handles all the processing required to illustrate entities and relationships, using graphical representations associated with these entities and relations. Although the display module is a vital part of a complete monitoring system, it was not implemented, except for a collection of routines to display derived tuples as they were generated. The coupling between the relational model used for monitoring and the model used for graphics is quite interesting, and is under active investigation.

The *relational database* component is responsible for the storage and retrieval of relations. This component functions as a "backing store" for the core module. An interface to the Ingres relational database system [Stonebraker *et al.* 76] has been designed, and is currently being implemented. However, there are still several unresolved issues in the coupling of the monitor core and Ingres, principally in the representation of object names (forming another name space, c.f., section 5.5.1) and the interface with the existing Ingres retrieval mechanisms. Since functioning implementations of neither the display module nor the relational database system currently exist, these modules will not be discussed further.

The *monitor core* is itself comprised of three modules (see Figure 8-2). The *Remote Accountant* handles the Ethernet protocol, sending tuples to the update network and sending commands to the resident monitor. The TQuel compiler and update network were introduced in earlier chapters.

The process decomposition actually implemented is similar to that described above, but not identical (see Figure 8-3). The TQuel parser has been extracted, and the remainder of the TQuel compiler merged with the update network. The parser was derived from the parser used in Ingres, and can thus benefit from the functions provided by the Ingres terminal handler, particularly the extensive macro facilities. The rest of the compiler should be separated from the update network to allow the monitor to process user commands and event records concurrently; they were combined here purely for ease in implementation. The processes on the Vax communicate using interprocess communication, or IPC [Rashid 80b]. IPC supports communication via structures messages between processes on the same or on different machines. IPC was not used for messages between the Vax and Cm* because it is not supported on Cm*.

The components of StarMon (the StarOS resident monitor) and Medic (the Medusa resident monitor) are illustrated in Figures 8-3b and 8-3c, respectively. The parser and the remote accountant were implemented in C [Kernighan&Ritchie 78], and the TQuel compiler and update network were implemented in FranzLisp [Foderaro 80]. The processes comprising the resident monitors are written in Bliss/11 [Wulf *et al.* 75b], a high level system programming language supported by both StarOS and Medusa.

The remaining sections of this chapter will focus on each of the components shown in

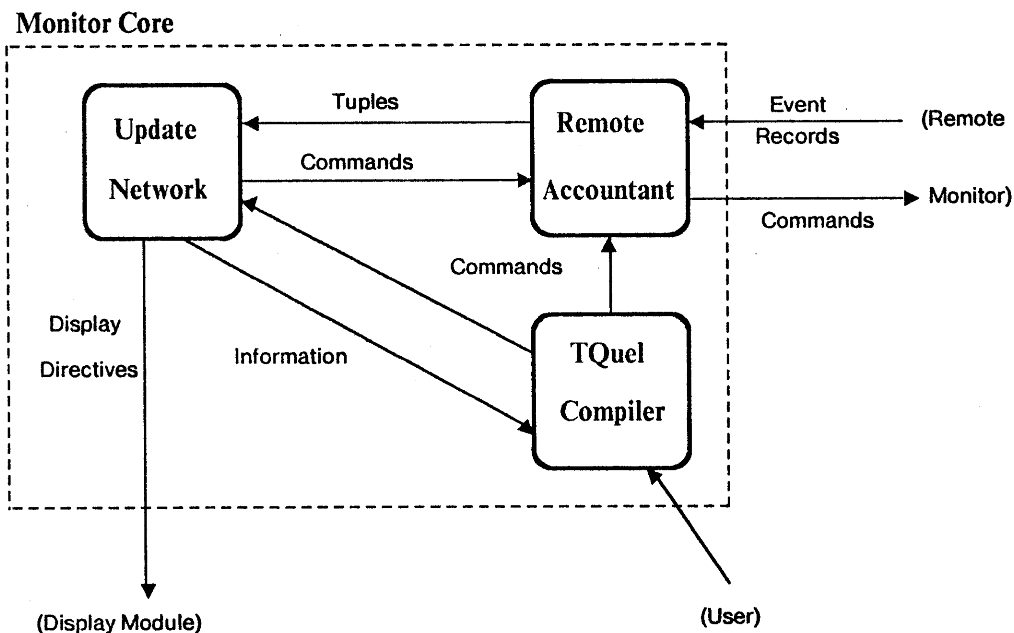

Figure 8-2: Components of the Monitor Core

Figure 8-3. The specification of sensors will be discussed initially, followed by an overview of the resident monitors. Finally, the implementation of the modules of the monitor core will be examined.

8.2. Sensor Specification

During the initial development of the low level event collection mechanism, it became apparent that there were several procedural difficulties in the placement of sensors in the various operating systems. These difficulties are compounded when general users start specifying sensors of their own. One difficulty was the that the sensor routine (which stores the event records) was becoming quite cumbersome to use. One design required six parameters for the simplest sensor, with additional parameters for each user-defined domain! Since the sensors were to be placed throughout the operating system, there was little room for error in the specification of these parameters.

A second problem was the assignment of event numbers: an incorrect event number in a sensor routine would result in the absence of event records of that type--a situation that might

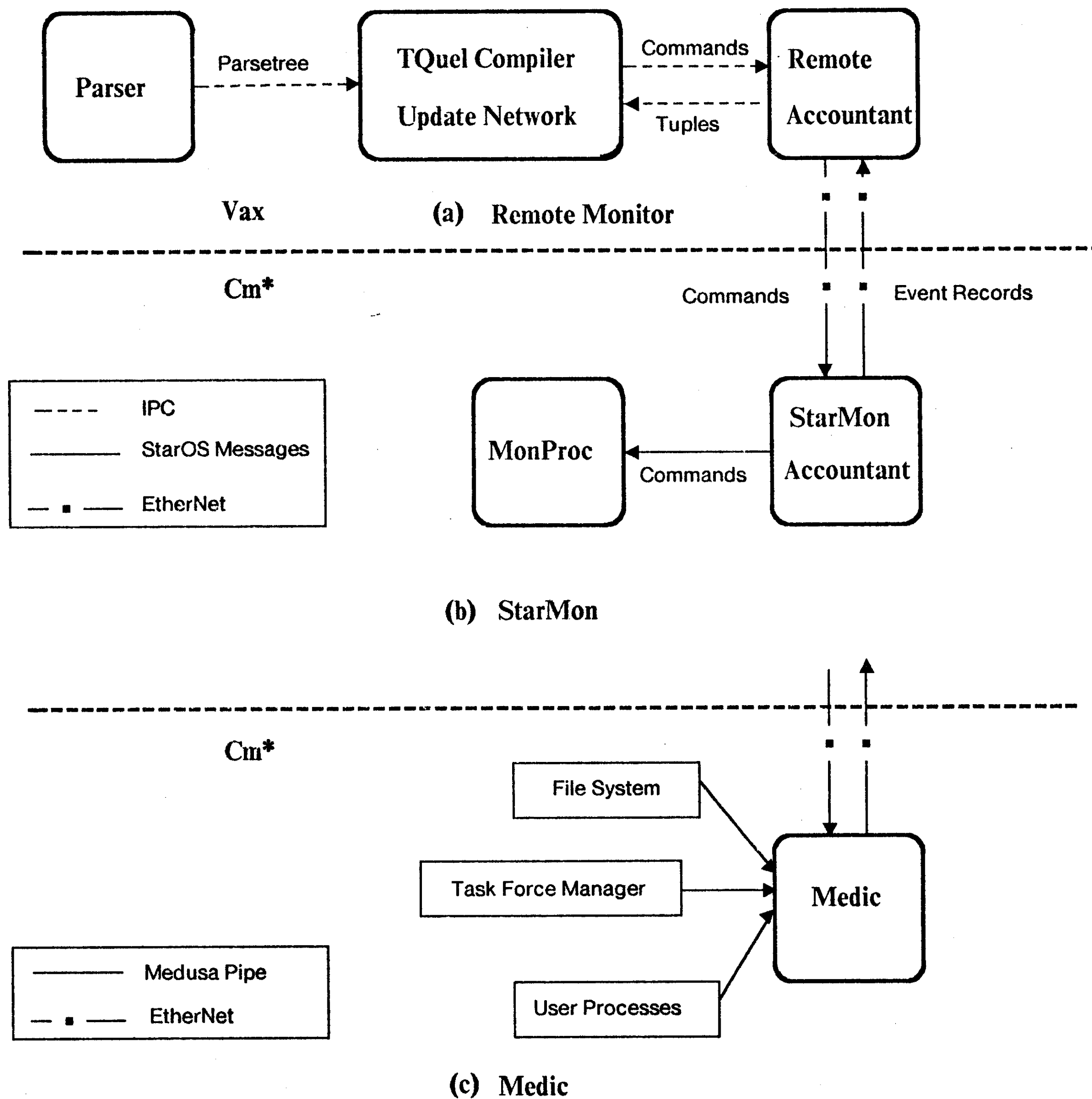

Figure 8-3: Structure of the Monitor As Implemented

be difficult to detect by the user interacting with the remote monitor. Two sensors with the same event number would cause havoc within the remote monitor since the monitor would interpret the user-defined domains of some of the event records incorrectly. Assigning unique event numbers to the many projects, programs, and versions of user applications and operating system was organizationally unmanageable.

A third problem was maintaining consistency between the remote monitor's view of the world and the world as it actually is. This is especially true during the early development of the monitor, when the collection of sensors inside the operating system, and the various attributes of those sensors, was changing frequently (this situation will continue to exist as long as there is active development of system software). Finally, all these problems are exacerbated by the sheer number of sensors: an operating system might contain several hundred sensors when it has been fully instrumented. The task of ensuring that all of these sensors, which are distributed across many source files, many users, and many versions of each program, are correct and consistent, both between each other and with the data structures within the remote monitor, is unmanageable if it remains a manual one.

In general, the less the system programmer or user has to specify, the less that can go wrong with the specification. In a best-case scenario, the user would specify the interesting events and indicate where the sensors for these events were to be placed in the code. The sensor would be produced automatically from the specifications, and would be as efficient as one crafted by hand. When the program was run, the event types generated by these sensors would automatically be defined as relations with the full query language available for manipulating events generated by the sensor. In addition, enabling and disabling of the sensors in the user's program would be handled automatically as a side effect of evaluating queries referencing these relations.

8.2.1. Sensor Description Files

The solution, described in this section, is to create a database, called the *sensor description file*, or *SDF*, containing information on the sensors defined in a given taskforce. A *taskforce* is a collection of processes cooperating to perform a particular function [Jones&Schwans 79]). Task forces roughly correspond to the concept of type managers as used in section 5.1. The sensor description allows the above scenario to be realized in its entirety: placing sensors in a program involves merely creating a sensor description file consisting of a few lines per event type and object type, adding the sensors to the program (one line per sensor), and running a program (called the *description file preprocessor*, or *DFPre*) with the SDF as input. All of the details are automatically taken care of by DFPre, in collusion with the resident and remote monitors and the Bliss compiler.

A sensor description file consists of a set of *objects* partitioned into *classes*. Each object is associated with a set of class-dependent *attributes*. There can be one or more *values* for each attribute, and some attributes can have objects as values. The syntax follows this description quite closely (see Figure 8-4).

As an example, the following description file includes an event object with three attributes, one having domain objects as values (this event class is a fragment of an actual SDF, reproduced in full in Figure E-1):

```
<description file>      ::= <objects>
<objects>               ::= <object> | <object> <objects>
<object>                ::= ( <class> <attribute list> )

<attribute list>        ::= <attribute> | <attribute> <attribute list>
<attribute>             ::= ( <attribute name> <attribute values> )

<attribute values>      ::= <attribute value> | <attribute value> <attribute values>
<attribute value>       ::= <object> | <object name> | <atom> | <integer> | <string> | <list>

<class>                 ::= <atom>
<object name>           ::= <atom>
<attribute name>        ::= <atom>

<atom>                  ::= <user defined name>
```

Figure 8-4: Sensor Description File Syntax

```
(event   (name Iteration)
         (timestamp true)
         (domains (domain (name iternum)
                          (type integer)
                  )
)
```

An SDF describes the sensors defined in a given taskforce. Since the operating system is itself a taskforce (or collection of taskforces), one SDF specifies the sensors embedded in the operating system. The **Taskforce** class includes attributes which hold for the task force as a whole. The **ObjectType** class describes the objects which can be monitored by sensors defined in the SDF. The **SensorProcess** class contains those attributes relevant to a particular process (i.e., a type manager). The **Event** class contains most of the attributes, including the following:

Location	the sensor process containing the sensor for this event;
Object	the object type this event refers to;
Timestamp	whether timestamps are to be included in the event record;
MinorType	how the event is to be triggered;

Domains the domains (optional user defined values) included in the event.

The **Domain** class includes attributes relating to each domain. Detailed descriptions of the various classes and attributes may be found in [Snodgrass 82].

8.2.2. The Description File Preprocessor

The *description file preprocessor* (DFPre) reads in a description, performs syntactic and semantic checking, and outputs several files containing information derived from the input file. The position of DFPre in the program development process is illustrated in Figure 8-5. In this figure, files are in boxes and programs are in ovals. The files provided by the user are marked with an asterisk.

The DFS (DeFinitionS) files (called "require" or "include" files) are processed by the Bliss compiler prior to reading the user's program. They contain macro definitions generated from the SDF. For example, each event class results in the definition of a sensor macro. If the SDF contained the event class shown above, then the IterationSensor macro, with one parameter (IterNum) would be defined. To place a sensor for this event, the user would simply put the line

```
IterationSensor(ThisIterationNumber)
```

in the code. ThisIterationNumber is a variable containing the current iteration number. The DFS files contain virtually all the details necessary for the monitor to interact with this sensor.

DFPre also assigns an event number to each event class defined in the SDF. It is important to note that event numbers are unique only within an SDF (and thus, within the task force associated with the SDF). There are always several taskforces executing concurrently on Cm*, and each one has, for instance, an event numbered one. The mechanism used by the remote monitor to disambiguate these events will be discussed in section 8.5.

The remote description (RemDescr) is a specially formatted file containing the information in the SDF of use to the remote monitor. This file is assembled and loaded with the user's program. When the resident monitor encounters the taskforce, it ships the remote description to the remote monitor as a series of event records. This operation is implemented as code in the resident monitor containing sampled sensors. When the operation completes, the remote monitor is aware of the events, sensor processes, and object types defined in the task force, and knows how to enable and disable these events. The most important aspect of the remote description is that it resides *with* the program containing the sensors delineated by the description; hence, consistency is guaranteed.

This mechanism also works for the SDF(s) associated with the operating system. Initially, the remote monitor knows of no events, sensor processes, or object types. The resident

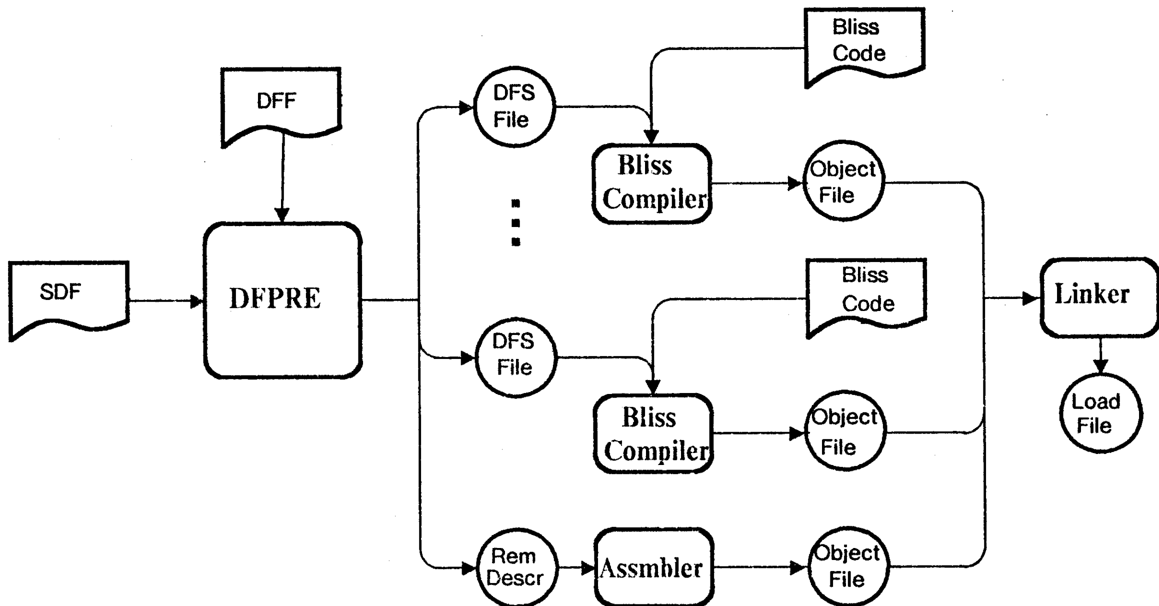

Figure 8-5: The position of DFPre in the program development process.

monitor first establishes contact with the remote monitor, then sends over the remote description of the operating system, thereby defining the events contained in the operating system. Since the resident monitor is part of the operating system, the sensors contained in the resident monitor are handled automatically by this procedure.

In addition to rectifying the problems introduced at the beginning of this section, using SDF's has several other advantages. Since DFPre has detailed information on the structure of each event, the code for that sensor can be tailored precisely to that event (the overhead of sensors produced by DFPre is quite low, as will be seen shortly). DFPre can also collect aggregate information concerning, for instance, all events located in a particular process, in order to perform global optimization. The data structures for each object type can also be configured quite precisely. Bootstrapping the description of sensors in the resident monitor itself is possible by predeclaring in the remote monitor only a few central sensors, primarily those that send the sensor description information to the remote monitor[22]. And finally, the information in the remote description can be used to compensate for resources consumed

[22]The current implementation uses only 4 predeclared sensors; a full implementation might require as many as a dozen.

during event collection, since the remote monitor will know what processing was involved in storing each event record.

A disadvantage of SDF's is that they are rooted in a compile-load-execute paradigm. The sensors must be specified at compile-time by the user. It would be desirable to allow sensors to be inserted dynamically, for example, by a debugger or by the monitor, much as breakpoints are inserted. Note that, by using the techniques developed for the Integrated Programming Environment [Habermann *et al.* 81], one can have the efficiency advantages of a compiler with the flexibility of an interpreter.

Although most attributes for the various classes are system-independent, a few additional, system-dependent attributes (and classes) are necessary. The allowable classes and attributes are defined in a *description file format (DFF) file*, input along with the SDF by DFPre. The motivation for a DFF file is similar to that for an SDF: manual specification is simply too difficult and error-prone. A DFF file is similar to an SDF file; the syntax is identical, yet the classes and attributes are interpreted by DFPre. The DFF file specifies

- the attributes to be placed in the remote description;
- the allowable attributes for each class;
- the allowable values (types) for each attribute;
- the default (if defined) for each attribute; and
- other system-dependent information.

Currently, four DFF files have been developed, including one for StarOS SDF's (see Figure E-4 for the full listing), and one for Medusa SDF's. Changing a particular aspect of SDF's in general, such as the default for a particular attribute, merely involves changing the DFF file.

This section has described how sensors are specified, and how this specification, the SDF, is manipulated by the monitor. The next section will examine the design of the resident monitor, including a detailed analysis of the performance of the sensors generated by DFPre.

8.3. The Resident Monitor

Chapter 5 presented a general low level data collection mechanism which could be applied to the various levels of abstraction present in a computer system. In order to partially validate the efficacy of this approach, the mechanism was implemented to monitor interactions at the process-process and process-operating system level. This particular grain was chosen for several reasons. Single process and single language monitoring tools have existed for quite some time, while multi-process monitoring has little prior experience to build upon. Also, it is not necessary when monitoring at this level to construct new hardware devices or program in microcode, activities which tend to divert attention from the fundamental issues. Finally, interactions at this level are the most complex, taxing the monitor's information retrieval and knowledge representation facilities to the maximum degree.

As was mentioned previously, two resident monitors on different operating systems were implemented: *StarMon*, on StarOS and *Medic*, on Medusa [Highnam 81]. Although both operating systems are quite similar in the abstractions they support, there are significant differences in the primitive operations provided by each system; these differences are reflected in the structure of the resident monitors. This section will discuss the StarOS implementation in detail, then examine the differences between StarMon and Medic.

8.3.1. StarMon: General Structure

The structure of StarMon is shown in Figure 8-3b. The *StarMon Accountant* maintains the Ethernet protocol with the remote monitor, in addition to collecting event records to be sent to the remote monitor. Most commands arriving from the remote monitor are simply channeled to *MonProc* through a StarOS mailbox. MonProc is responsible for performing the indicated operations. This partitioning was necessary both for performance and reliability reasons. Due to limited main memory resorces for temporary storage of event records, the StarMon Accountant must continually scan these storage areas and remove event records accumulating there. Any abnormal delays may result in the storage areas overflowing, with a subsequent loss of event records. Rather than having the StarMon Accountant risk losing event records while executing a lengthy command, these commands are instead leisurely performed by MonProc, with few real-time constraints. In addition, an error might occur in executing a command. Since MonProc executes all potentially dangerous commands, it can simply reinitialize itself when an error occurs. The StarMon Accountant does not have such freedom, since it must maintain the Ethernet protocol with the remote monitor. Hence, it executes only the simplest and fastest commands.

Multiple instantiations of MonProc are allowed if the command rate is too high for one process to handle. Multiple instantiations of the StarMon Accountant could be accommodated with minor changes to it and to the Remote Accountant. Such a change assumes

that the Remote Accountant is fast enough to handle several Ethernet connections simultaneously.

8.3.2. Sensor Performance Measurements

Small is beautiful.

-- Schumacher's dictum

The various data structures and algorithms employed in StarMon are discussed in Appendix D. This appendix shows that the space overhead for including sensors in a process is between .5 and 4%, depending on the number of sensors installed, and less than 2% for allowing an arbitrary StarOS object to be monitored. The memory overhead for temporary event record storage is between 1 and 7%, depending on the variability of the event generation process[23].

The performance of StarMon sensors was analyzed along two dimensions: implementation (procedure call, inline code, microcode) and information collected by the sensor. Three implementation versions of StarOS sensors were analyzed. The first version consisted of inline code which tested whether the event was enabled, and, if so, called a procedure to construct and store the event record. The second version consisted totally of inline code. Both versions were implemented using existing microcoded operations for efficiency. The third version was designed to be implemented fully in microcode. Its performance was estimated using computed memory reference counts are extrapolating from previously measured microcoded operations. Details on the microcoded version may be found in appendix D.5.

Measurements for these versions for five sample sensors is shown in Table 8-1. The sensor description file for these sensors (among others also tested) is shown in Figure D-7. These sensors differ primarily in the amount and type of information they store in the event record. Sensor A (Shortest in Figure D-7) is the minimal sensor: no timestamp, no domains, always enabled. An example is

```
FileRead (File)
```

which indicates which file was read, but not the time the file was read or any other information about the read operation. Sensor B (Short) includes the timestamp and one user-defined integer domain. An example would b

```
BlockRead (File, BlockNumber)
```

Sensor C (Medium) includes a user-defined, double precision integer domain. Sensor D

[23] If events were generated at a fixed rate, only a small amount of internal storage would be necessary. Larger amounts of storage are necessary if many event records are generated quickly, even if this occurs only occasionally.

Sensor	Procedure call		InLine			Microcoded		
	Size	Time	Size	Time		Size	Time*	
	words	microsecs	words	microsecs	CALL	words	microsecs	CALL
A	29	1850	41	585	5.9	7	250	2.5
B	36	2025	60	725	7.3	12	350	3.5
C	42	2150	62	765	7.7	15	400	4.0
D	44	2330	87	1395	14.0	18	560	5.6
Iteration	--	--	96	970*	9.7*	12	350	3.5

* Estimated

Table 8-1: Performance Measures for StarOS Sensors

(Long) includes a variable-length character string domain (the time values are for storing a 15 character string). The last sensor is the one specified in the extended example presented in appendix E. It is similar to sensor B, with more overhead due to tighter memory constraints (analogous to a shortage of high speed registers). For all sensors, if the event was disabled, the sensor took 165 microseconds, equivalent to 23 store operations. If no events were enabled for the process or object, the sensor took only 15 microseconds (or 2 store operations), representing the overhead of installing a sensor and never using it. The microcoded version would take approximately 90 microseconds if the event was disabled, and 40 microseconds if no events at all were enabled. The reason the microcoded version in this one case is significantly slower than the non-microcoded implementation is that there is a fixed overhead of about 30 microseconds to invoke a microcode instruction.

Three surprising results were obtained from these measurements. The first is that inline macro expansion, as opposed to calling a procedure to store the monitoring data, is *always* appropriate. Calling a procedure is approximately three times slower than inline code. Procedures are always slower than equivalent inline code, so the time penalty was expected. However, the inline code to perform the data collection is only slightly larger than the single procedure call! Since the procedure itself is 350 words long, inline expansion requires less space than procedure calls if there are less than 20 or so sensors in the process. The reason the inline expansion is so fast and space efficient is that the code for each sensor is highly customized (by DFPre) to the exact specification of the sensor, as found in the SDF. For the procedure call version, the various alternatives must be communicated to the procedure as parameters, requiring additional space and time. An intermediate alternative would be to have, say, three or four generic procedures, each specialized to a set of parameters, with DFPre selecting the appropriate version for each sensor. At this point, the potential gains start decreasing and one is tempted to simply apply techniques such as generic procedure expansion to the entire program, rather than to only the sensor routines [Saunders 79, Rosenberg 83].

The second surprise is that inline expansion is close in time to a fully microcoded version. Overall the microcoded version was only 50-60% faster than the mixed version. On the other hand, it would take an estimated two months for an experienced programmer familiar with Cm* microcoding to implement and test the microcoded version [Vegdahl 82]. In addition, the microcoded version would increase KMap contention, which could slow down concurrent memory references or other microcoded operations. There are, however, two distinct benefits to the microcoded version: it is significantly smaller than the inline version, and it is not susceptible to memory addressing constraints. For example, the microcoded version of the Iteration sensor is identical in time to sensor B, which has few memory addressing constraints. The inline version of the Iteration sensor is significantly slower and larger than the inline version of sensor B.

Most efforts at microcoding result in at least an order of magnitude speedup. The reason why microcoding seems not to be a fruitful activity in this case is, again, the careful coding of the mixed version, plus the fact that the mixed version relies heavily on existing microcoded operations (see appendix D.4 for details). The microcoded sensor must be general, whereas the mixed version is specific to the sensor. Hence, these results indicate that microcoding effort should be invested in other parts of the operating system first, where greater gains are possible.

The third surprise resides in the CALL characterization of execution time. An execution time in microseconds is only valid in a comparative analysis on the same machine, was done as above. The CALL column specifies the execution time as the equivalent number of procedure calls (at approximately 100 microseconds each). This characterization of execution time is roughly the ratio of the effort necessary to store the event and the effort to call a procedure, providing an estimation of how fast a similar implementation would be on a machine other than Cm*. The measurements indicate that the monitoring grain is between 6 and 10 times coarser than a procedure call. Put another way, the monitoring grain for this data collection mechanism is larger than a procedure call, but perhaps equal to a procedure that does something interesting, in turn calling other procedures. It should be noted that this comparison is still somewhat machine dependent, since it includes assumptions concerning the overhead of microcode invocation (rather high in Cm*), procedure call overhead (also rather high), and relative efficiencies of microcode versus machine instructions.

The third result also verifies an assumption made earlier, namely, that sophisticated filtering techniques are necessary to greatly reduce the number of event records generated. As a rough estimate, if monitoring with no filtering (all sensors are always enabled) incurs an execution time overhead of 20%, then the 50 processors in Cm* could generate 10,000 event records per second, most of which would be immediately rejected in the update network. If filtering reduced the overhead to 1%, the 50 processors would generate a much more manageable 500 event records per second, most of which would be used by the update network. As will be demonstrated later, it is possible for the other components of the monitor to handle 500 event records per second, but certainly not 20 times that quantity.

8.3.3. Medic

The structure of Medic, the resident monitor for Medusa, is shown in Figure 8-3c. Medic consists of one process composed of a collection of coroutines [Highnam 81]. The sensors are quite similar in StarMon and Medic: the space requirements and execution times are comparable, and the implementations follow each other quite closely. Medic supports an identical protocol (see below), including initially sending the operating system's SDF and executing all the commands. From the previous discussion, the single process structure would appear to be less efficient and robust compared with StarMon. More explicit statements concerning the relative efficiency and robustness must wait until Medic itself is instrumented; Medic has been designed and implemented and is now being tested. However, the existence of two resident monitors partially demonstrates independence at the level of the monitored operating system.

8.4. The Ethernet Protocol

The event records are initially generated by the sensors and placed in temporary storage areas, waiting to be picked up by the resident monitor and sent to the remote monitor via the Ethernet. The protocol [Highnam&Snodgrass 81] is a variant of the EFTP (Ethernet File Transfer Protocol) [Shoch 79], simulating a transmission from the remote monitor (the host) to the resident monitor (the slave). This protocol may be thought of as a modified transport protocol using the Pup protocol as the packet layer of the communications hierarchy [Boggs *et al.* 80, Davies *et al.* 79]. The commands are sent in the data packets and the event records are placed in the acknowledgement packets. The protocol uses checksums, timeouts, and packet retransmission for reliability. Since the resident monitor is a slave in the protocol, it must wait for the remote monitor to send a packet before it can respond with an acknowledgement containing data. Hence the remote monitor must occasionally send packets even if there are no commands to be sent[24]. The resident monitor indicates in every acknowledgement the amount of buffer space it has free, allowing the remote monitor to adjust the packet transmission rate accordingly.

The following commands are supported:

Adjust Object — either enable or disable an event;

Checkpoint — generate a checkpoint;

Read Entry — ask the resident monitor for some information to be sent back as a Report data record;

[24] Of course, the minimum packet frequency depends on the event generation rate.

Write Entry send the resident monitor some information;

End indicate the last command in a packet.

The following variable length data records are supported:

Event Record generated by a sensor:

Report generated in response to a ReadEntry command;

Check a checkpoint;

Error report an error;

New Name report that an object has been garbage collected; and

Last Record indicate the last data record in a packet.

Given the record and packet sizes and observed transmission rates for the standard EFTP, a rough maximum event record transmission rate is 600 event records per second, comparable to the maximum event generation rate given in section 8.3.2. The actual transmission rate has not been measured; this estimate provides an indication of the *possible* performance of this protocol.

8.5. The Remote Accountant

The remote accountant handles the details of the Ethernet protocol, including determining which primitive relation each event record is associated with, unpacking the fields of the event record into a tuple, and generating remote names from the internal names present in the event record.

The field unpacking is straightforward: the monitor core informs the remote accountant of the number and types of domains present in each primitive relation (this information is extracted from the SDF sent by the resident monitor). The fields are formatted for efficient handling by the update network.

The mapping from internal names to remote names is also not difficult. The remote accountant stores, for each internal name, the time period when each <epoch> was valid (see section 5.5.1). A new epoch for an internal name begins when the object with that name is garbage collected; the remote accountant is informed of this occurrence through a New Name data record (see section 8.4). The timestamp is used to determine which <epoch> is

appropriate for each internal name. Note that the checkpoint tuples can be used to keep the number of entries low, thereby reducing search time during this mapping. Although not currently done, this process could also be used for object names appearing as values in user-defined domains.

The remote accountant uses a variety of information to determine the associated primitive relation for each incoming event record. Each primitive relation is assigned an integer called the *unique event number*. Each taskforce is also assigned an integer called the *unique tf number*. Both of these assignments are made by the update network as it processes the event records encoding the remote descriptions of the SDF's. In addition, as the identity of taskforce components is ascertained by the update network, it sends the remote accountant pairs of remote names and unique tf numbers.

When an event record arrives, the remote monitor first examines a specified bit in the event record to determine if the event was an internal or external event. An *internal* event is associated with the process generating the event; an *external* event is associated with the object the operation generating the event was performed on (see section 5.2). The remote accountant then extracts either the object name or the sensor name, if the event was an external or internal event, respectively (both names may be present in the event record). This name is used to determine the task force (unique tf number) associated with this event. Finally, using the unique tf number, the internal Boolean, and the local event number in the event record (assigned by DFPre; see section 8.2.2), the unique event number is determined. It is this value which identifies destination(s) (access nodes) in the update network the tuple is sent to.

The design of the remote monitor assumes that only one user is using the monitor, and therefore that there is only one update network to route incoming event records to. There are no fundamental difficulties envisioned with having multiple users.

8.6. The Update Network

The update network, as described in chapter 6, accepts tuples from the remote accountant and produces derived tuples, either to be stored in the relational database or displayed graphically by the display module. The update network as viewed by the TQuel compiler is specified by the three operations create, link, and unlink, and by the defined generic access and operator nodes, allowing great flexibility in the implementation. For instance, when an access node is created, the monitor updates several data structures in order to ensure that the tuples flowing from the remote monitor are routed efficiently to the correct nodes. Tailoring of the operator nodes based on the instantiation parameters occurs when the nodes are created. Finally, the link operation performs quite different actions depending on the types (and sides) of the nodes it is connecting.

There are two primary approaches to implementing the update network: either the connectivity graph is stored in a data structure and the movement of tuples simulated by an interpreter which invokes the operator nodes, or the network is compiled into code which executes directly. The trade-offs are similar to the implementation of other language systems: interpreters are more flexible, but compiled code is more efficient. An update network interpreter was implemented; from the experience gained in that effort, an update network compiler was designed. Performance measurements were taken to determine the efficiency of the implementation.

This section will first examine the representation of tuples, followed by discussions of two versions of the update network, an interpreted version and a compiled version.

8.6.1. Tuple Representation

Tuples are represented by a list of domains. All tuples have three implicit domains: the unique event number of the tuple, the *class* of the tuple, and the time value for the tuple, consisting of one or two timestamps. The following classes are included:

Event — the tuple represents the occurrence of an event; the time value is a single timestamp;

Period — the tuple represents a relationship which existed for a period of time; the time value is the pair <begin timestamp, end timestamp>;

StartPeriod — the tuple represents the start of a relationship, with one timestamp;

StopPeriod — the tuple indicates that the relationship no longer holds, with one timestamp; and

Sample — the tuple encodes a relationship which was true at the time specified by the time value, a single timestamp.

A *timestamp* is itself the pair of times <start uncertain, start certain>, allowing for "fuzzy events" (see section 4.5). The primitive relations have little indeterminacy, on the order of tens of microseconds (depending on how variable the execution time of the sensor is). Derived relations, especially those derived from sampled relations, have much greater indeterminacy.

The explicit domains (the object, process, and user-defined domains) are typed, and have a type-dependent structure. Since the tuples flowing on a particular arc of the network compose a relation, they are guaranteed to have an identical structure. The possible domain types are integer, string, remote name, and temporal. A remote name (see section 5.5.1)

specifies an object which exists (or did exist) in the system being monitored. A temporal domain is the ratio of two linear functions of time, and is represented by 4 values specifying the slope and the intercept (relative to the start uncertain time of the start timestamp) for the numerator and the denominator. An example of a temporal domain is the domain calculated by the Duration operation (see section 3.6); this domain consists of (1, 0, 0, 1), representing $(1 \times t + 0) / (0 \times t + 1)$, or t, implying a value starting at 0 and increasing linearly with time. Given these values, it is easy to calculate the value of the domain at any point in time between the start and stop time, and to incorporate fuzziness into the value of the domain. A linear function of time (requiring only two values instead of four) is sufficient for most purposes; the additional values are necessary primarily for cumulative aggregate operators (see section 3.6).

8.6.2. Interpretation and Compilation

The interpreted version of the update network is organized around a collection of data structures representing the access and operator nodes. The primary mechanism is the association of functions with each node which are invoked at specific points during each operation involving the node. For instance, associated with each generic node are functions to be invoked when a node is instantiated from this generic node, and when the instantiated node is linked *to* another node, linked *from* another node, unlinked, or sent a tuple to its left, right, or control input. In the last three cases, if the function returns a tuple or list of tuples, those tuples are automatically sent to the nodes connected to this node. Also included in the data structure of each node are the instantiation parameters (names or values, depending on whether the node was a generic or instantiated node, respectively), temporary space for values to be stored across invocations of the associated functions, and debugging information. The functions associated with the node are given access to these fields by including the node as one of the parameters to the function.

Organizing the update network as a collection of data structures with associated functions has several advantages over a more standard, monolithic implementation. The create, link, and unlink operations themselves are small; instead of performing the operation directly, they merely check for the presence of a appropriate function, and if present, then invoke the function with the involved node as a parameter. Adding a new generic node is straightforward and does not require changes to any existing code. Such *object-oriented* programming has found wide use in programming languages [Dahl *et al.* 67, Ingalls 78], artificial intelligence systems [Winston&Horn 81, Charniak *et al.* 80], operating systems [Jones *et al.* 78, Wulf *et al.* 81, Ousterhout *et al.* 80, Kahn *et al.* 81], and command languages [Snodgrass 83], has served well in this application.

Since the algorithms in the nodes are embedded in the associated functions, the primary task of the update network is ensuring that the tuples get to the correct nodes in the correct

order. Breadth-first scheduling (see section 7.2.4) was implemented, since it was felt at the time that allowing the tuple streams to get out of synchronization would cause the update network to generate incorrect results. After carefully considering synchronization and temporal order, it became apparent that breadth-first scheduling alone would not guarantee correctness. Hence, in retrospect, it appears that depth-first scheduling is the preferable approach. As will be seen below, depth-first scheduling was implemented in the compiled version due to its increased efficiency.

The TQuel compiler as implemented generates correct update networks, but does not include most of the optimization strategies discussed in section 7.2. Although the interpreted version was flexible and relatively easy to implement, it had one major drawback: it was slow. As the measurements to be presented below indicate, the maximum tuple rate achieved, even with the optimizations, was less than 25 tuples per second. Given the estimates derived earlier in this chapter, this rate is about an order of magnitude too low. To achieve such a speedup, it was necessary to abandon some of the flexibility afforded by the interpreter.

The update network compiler, as opposed to the *TQuel* compiler, which produces an update network from a TQuel query, translates a sequence of create and link operations (i.e., the update network) into a collection of Lisp functions, which are then compiled by the Lisp compiler[25].

The update network compiler was designed but not implemented. However, the techniques involved in compiling update networks are commonly found in standard compilers. Details of possible optimizations may be found in appendix F. Construction of an update network compiler should be a straightforward task not requiring any new breakthroughs.

8.6.3. Update Network Performance

Table 8-2 compares the various implementation techniques. Three sets of measurements were taken; one with the update network generated by the existing compiler, one with the update network optimized by hand, using only strategies that could be readily implemented, and one with Lisp functions generated by hand from the optimized update network, again using only strategies that could be readily implemented. The update networks all implement the queries given in section 3.7. It should be emphasized that the measurements only apply to this one set of queries, and may not be representative of queries in general. On the other hand, these queries are somewhat complex, involving several <tuple variable>s, <when clause>s, <where clause>s, <start clause>s, and <at clause>s, and several expressions. The measurements include the number of fires per input tuple (a *fire* is the invocation of an access

[25]Note that the update network compiler could also be implemented to generate C routines, which would then be compiled into assembly code.

or operator node), the execution time per fire, and the number of input tuples which could be absorbed each second. The details of these and additional measurements are given in appendix F; this section will summarize the results.

Version	Tuples Generated	Fires Per Input Tuple	Milliseconds Per Fire*	Input Tuples Per Second*
Initial	32	19.9	7.6	6.6
Transformed	32	9.2	4.7	15
Compiled	32	1.0	1.5	660

* On a dedicated Vax 11/780

Table 8-2: Performance Measures for the Update Network

Since all three update networks were correct, they generated identical output tuples for the same input tuples. For a set of 50 test input tuples, chosen to produce interesting results, there were 32 output tuples produced (see Table 8-2). All trials with actual input tuples resulted in few output tuples, since one process tended to get and stay ahead of the other process (recall that the queries investigate the interplay between the two processes). Hence, since each input tuple that results in an output tuple causes more node fires than an input tuple that gets eliminated during the processing, the measurements using the test input tuples are quite pessimistic for this update network.

In the non-transformed network, each input tuple resulted in almost 20 node fires, generally as a result of the selection operator nodes being after the Cartesian product nodes (again, see appendix F for details). The execution time per node went down by 40% from the original to the optimized update network, primarily due to the more favorable temporal order. These two reductions cooperatively increase the number of tuples processed per second by an impressive factor of 5.

An even larger increase (a factor of 40) occurs with the update network is compiled. There is only 1 fire per tuple in the compiled version because each update network in this approach is in effect a highly specialized operator node (internal function calls were not counted). In fact, the optimizations on the original update network, coupled with conversion of the update network into Lisp, and then into assembly language, result in an improvement of two orders of magnitude. The number of input tuples processed per second is comparable to previous tuple rates computed for the resident monitor and the Ethernet protocol.

8.6.4. Relationship to Data Flow

The update network as constructed by these operation out of the nodes described above resembles a data-flow program. Data flow programs, in their graphical representation, consist of computational nodes connected in a graph structure. Values, in the form of tokens, flow over the arcs during the computation. Data flow programs meet three criteria [Aggerwala&Arvind 82, Davis&Keller 82]:

- freedom from side-effects,
- data availability firing rule, and
- lack of history sensitivity in procedures.

The first and second criteria follow by definition for the update network; there are *no* global data structures and a node fires when a tuple appears on its input arc. One can argue that the third criteria doesn't hold; one example is the join node, which concatenates an input tuple from the left arc with all previous input tuples from the right arc, then applies a predicate to the result. Although there is history sensitivity in the order of the output tuples, the same tuples are eventually generated independent of the order of the input tuples. Since order is semantically irrelevant, the update network is in fact a data flow program.

Since each update network is in fact a data flow program, it follows that techniques and hardware developed for executing data flow programs would apply directly to executing update networks [Dennis 80]. A variation on this idea may be found in [Boral&DeWitt 81, Boral&DeWitt 82], where a dataflow implementation of the (non-temporal) relational algebra at the granularity of pages, rather than tuples, is described.

8.7. A Step Back

Behind the implementation described in this chapter, and at the core of the research, is an iterative process for building quality software:

1. Determine the "correct" conceptualization of the program to be implemented.
2. Develop an interface (user-program or program-program) that adheres as closely as possible to that conceptualization.
3. Test the conceptualization with a highly flexible, probably inefficient implementation.

4. Apply all the available information to a particular instance of the problem in order to efficiently perform the desired action, trading generality for efficiency.

5. Demonstrate as formally as possible the semantic integrity of the translation from general to specific.

6. Use available technology to the greatest extent possible at all stages.

This process certainly contains nothing new: in some form, it is at the root of high level languages, operating systems, many design methodologies, and successful software systems. What is new is the application of this process to yet another software system, a monitor for distributed systems. Three instances of this process as they occurred in the implementation will be outlined, followed by a succinct statement of the major results of the implementation.

Example 1:

1. Monitoring "should" be viewed as retrieving information from a dynamic relational database.

2. TQuel is a user interface that adheres to this viewpoint.

3. The relational calculus serves as a flexible yet inefficient "implementation".

4. A TQuel query can be translated into a relational algebraic expressions, and can then be optimized using a collection of transformations.

5. Each TQuel query was proven to have an equivalent tuple calculus expression, which can be proven to have an equivalent relational algebraic expression.

6. The available technology consisted of the mathematical basis for relational databases and the semantics of standard relational queries, as well as the Ingres [Stonebraker *et al.* 76] and yacc [Johnson 75] systems to aid in lexical analysis and parsing.

Example 2:

1. Sensors "should" be viewed as parameterized actions.

2. Sensor Description Files (SDF's) constitute a user interface for specifying the attributes of the user's sensors.

3. A sensor routine was implemented, and various approaches were investigated to determine how the event information could be communicated to the monitor.

4. DFPre converts an SDF event class into a highly parameterized macro call.

5. The sensor routine and inline macro share most of the code, ensuring that both implementations perform identical operations.

6. The available technology included the highly optimizing Bliss/11 compiler [Wulf *et al.* 75c], the concept of a linearized representation of a parse tree as developed in the PQCC project [Leverett *et al.* 80], and the use of frames as a data structure [Winston&Horn 81].

Example 3:

1. The target code for the TQuel compiler "should" be an update network.

2. The interface consists of the three operations create, link, and unlink, and the collection of generic access and operator nodes.

3. An interpreter-based implementation was developed to test the concept of an update network.

4. A compiler was designed to translate an update network into an efficient Lisp program.

5. The integrity of the translation process was partially validated by designing a compiler using many of the same data structures used by the interpreter.

6. The available technology included techniques for compiling production systems [Forgy 79], the Lisp compiler [Foderaro 80], and the specification of Bonsai, a tree transformation system used in the PQCC project [Leverett *et al.* 80].

8.8. Evaluation

Every honest calling, each walk of life, has its own elite, its own aristocracy based on excellence of performance.

-- *James Bryant Conant*

A chain is only as strong as its weakest link.

-- Proverb

This chapter has presented, in some detail, an implementation of a relational monitor. Although not all components were implemented--the display module, relational database interface, and update network compiler were only brought through the design stage--and the components that were implemented are incomplete, enough was implemented to assess the viability of a complete implementation.

The criteria given in chapter 2 for an effective monitor relating to implementation were efficiency and system-independence. System-independence is an ill-specified criterion. However, the existence of two resident monitors, running on rather different operating systems, at least indicates that some measure of system-independence has been achieved. Of course, the more resident monitors implemented, the higher the confidence in this assertion.

There are two aspects to the issue of efficiency: is the monitor as a whole efficient enough, and do the components have comparable efficiencies? The latter issue exposes bottlenecks where high efficiency of some of the components is wasted by one or more low efficiency components. Given the monitoring grain supported in the implementation, that of process-process and process-operating system interactions, the efficiency of the implementation is adequate. Section 8.3 showed that an overhead of 1% results in the generation of approximately 500 events per second; section 8.4 came up with an approximate event record transmission rate of 600 events per second; and section 8.6.3 indicated that a processing rate of over 600 input tuples per second was possible on a dedicated Vax. Thus, the Ethernet and remote monitor can essentially keep up with the 50 processors on Cm*. It is also fair to say that if the situation is changed somewhat; say, an additional 20 processors are added to Cm*, or the Ethernet or the Vax get loaded, then the monitor as realized here would *not* be able to keep up.

The point to be emphasized is that it is in fact possible to implement a monitor supporting the high level conceptual viewpoint of a collection of time-varying relations which can be manipulated by a temporal, non-procedural query language, with enough efficiency to monitor a large, complex distributed system. Although further measurements and a careful analysis of the performance of the various components are certainly needed, the issue now is, *how large* a distributed system can be monitored in this way, rather than, *can* a distributed system be monitored in this way. Hence,

> **Result:** With careful design and the use of previously developed concepts, techniques, and tools, it is possible to support a powerful, high-level conceptual interface to a distributed system monitor with an efficient, nicely structured implementation. The user does not have to specify how the data is collected, or worry about the details of collecting, formatting, and processing the event records.

IV. Conclusion and Appendices

The previous chapters have attempted to stay as abstract as possible while still presenting enough detail to show how the mechanisms work. For the curious, several appendices have been provided to tie up the loose ends.

Chapter 9
Conclusion

If a man will begin with certainties, he shall end in doubts; but if he will be content to begin with doubts he shall end in certainties.

-- *Francis Bacon*, in The Advancement of Learning

The thesis of this research is that monitoring distributed systems is fundamentally a knowledge representation and information processing task, and that the user interface and internal structure should be conceptualized in this way, that is, in the relational paradigm. Given this approach, many questions immediately come to mind, involving such issues as

- data collection;
- dynamic incremental updating of temporal relations;
- syntax and semantics of the query language;
- translating a query into a more convenient form; and
- implementing the proposed mechanisms.

The results generated from an examination of these issues are summarized in result statements at the end of each of the previous chapters.

Several specific results constitute the major contributions of this research:

- A temporal relational query language, TQuel, was developed by syntactically and semantically augmenting an existing query language and by incorporating previous work in synchronization (i.e., path expressions).
- A formal semantics was defined for the entire TQuel **Retrieve** statement. This semantics has several desirable properties:

- It reduces to the standard Quel semantics when the time domain is fixed at a particular time.

- It includes aggregate operators in a uniform fashion.

- It accommodates an arbitrary degree of indeterminacy.

- A low level data collection mechanism was presented. This mechanism, based on a strongly typed model of the environment, supports several dimensions of filtering, integrates sampling and tracing, and admits solutions to the problems of naming and time.

- Update networks were proposed to implement dynamic incremental updating of derived temporal relations. The network is composed of access nodes, which interface effectively with the resident monitor, and operator nodes, which carry out the desired computations.

- Several general techniques were developed to generate correct and efficient update networks from TQuel queries. These techniques include:

  - the translation of the query into an initial update network via the relational operator tree;

  - modifications to this network to ensure semantic correctness; and

  - the use of graph transformations, temporal order, and node scheduling to greatly increase the efficiency of the network.

  These techniques made extensive use of knowledge available to the monitor.

- Several problems concerning the specification of sensors were solved by using sensor description files. The user does not have to specify how the data is to be collected, or worry about the details of formatting or processing the event records. By using the information in the sensor description files and by careful use of available microcoded operations, highly optimized StarOS sensors were implemented.

- By compiling the update network, an increase of over an order of magnitude in execution time was obtained.

9.1. Surprises

Many of the results stated above were at least partially anticipated when this research was first proposed. However, there have been several interesting surprises along the way. These surprises will be presented as previously held assertions that turned out *not* be true in this case. Each assertion can be interpreted either as a statement that should have been recognized initially as being false[26], or as a generally valid observations that has less applicability than previously believed.

- Naming is not a problem when monitoring, because the sensors will in general have access to names of objects involved in the events the sensors are recording.

  As shown in section 5.5.1, there is in fact *no* mechanism for producing remote names which will satisfy all the required invariants. The solution adopted there was to slightly corrupt one of the invariants, resulting in a minor reduction in the ability of the monitor to collect information.

- Inserting sensors is a routine, yet time-consuming process.

  All past experimentation on Cm* has relied on manually inserted code to collect the information and, later, to analyze it. When the mechanism described in section 5.2 was adopted, sensors became more versatile, and consequently, more complex. Eventually, I realized that inserting sensors was becoming quite difficult procedurally, given sensors spread across many users, programs, and files (see section 8.2.1). Inserting sensors was no longer routine, and was even more time-consuming. The solution was to use automatic assistance via sensor description files, resulting in a routine, fast method for inserting sensors.

- Extending aggregate operators to include time should be easy, yet extending the semantics to incorporate an arbitrary degree of indeterminacy is very difficult.

  Specifying the semantics of aggregate operators on temporal relations turned out to be amazingly complex. At least seven (!) different interpretations of the temporal **avgc** operator were considered (see section 4.4.1). On the other hand, to accommodate indeterminacy required only two changes: adoption on a three-valued logic system and a straightforward redefinition of the **before** predicate (see section 4.5).

---

[26] In which case, I am merely admitting my prior ignorance.

- Implementing sensors in microcode, rather than in Bliss/11, a high level systems programming language, should result in a significant decrease in execution time and program space.

  The tracer routine in C.mmp, which performs a similar function to the StarOS sensor, was originally implemented in assembly language, and required nearly 140 instruction equivalents in execution time (350 microseconds on PDP-11's) [Wulf *et al.* 81]. Microcoding this operation reduced the execution time by a factor of 5 to 26 instruction equivalents (65 microseconds). Microcoding other instructions has resulted in reductions of one or more orders of magnitude. The experience with the StarOS sensors did not follow these expectations. The microcoded sensor was only about *twice* as fast as the implementation using a combination of Bliss/11 code and existing microcoded instructions; see section D.6 for the details.

- Inline expansion runs faster, but procedure calls take up less code space.

  With regard to StarOS sensors, inline expansion was faster *and* shorter (again, see section D.6).

- Processing temporal databases must be much slower than conventional databases, since they are so much larger (a conventional database contains only the slice of a temporal database currently valid.)

  By utilizing the techniques presented in section 7.2, it is possible to exploit the temporal order of incoming tuples. In this way, each new tuple affects only a small portion ot the database--that portion which was close in time to the incoming tuple. Information in the database that is removed in time from the incoming tuple is not affected by the addition of this tuple. This use of *temporal locality* effectively reduces the size of the database as seen by the update and retrieval algorithms, making a temporal database as efficient as a conventional database which stores only the most recent portion of the information in the temporal database.

9.2. Remaining Problems and Future Research

It is better to know some of the questions than all of the answers.

--James Thurber

The remaining problems provide excellent opportunities for future research. There are at least four directions the work reported here can be extended along. The most natural direction is to complete and extend the implementation. The second avenue for further work is to integrate the monitor with the other tools in the programming environment. The relational approach indicates another overlap, that of databases in general and temporal databases in particular. And finally, there is a need to pursue an area motivating this research, that of knowledge representation. These areas each suggest several more promising research topics.

The most striking need in the implementation is a display module.There has always been a strong connection between data structures (and thus data bases) and graphics. Leap, a language supporting an associative memory (a precursor to the relational model), was developed to aid in the implementation of graphical applications [Feldman&Rovner 69]. Recently, several graphics systems employing a relational data base have been implemented [Becerril *et al.* 79, Williams 74, Herot 80].

The techniques developed in these systems suggest that a graphical interface for a monitor designed around the relational model would be possible. However, there are still several interesting problems in designing the display module, including representing indeterminacy, controlling the display rate, utilizing the available visual modalities (color, blink rate, absolute and relative position, shape, movement) of the display device, and specifying these attributes in TQuel. Another area for improvement in the implementation is the interface with Ingres. Also, essentially all of the modules require significant effort before the monitor can be used by unsympathetic users.

The monitor was implemented as an autonomous unit, interacting minimally with the other software development tools comprising the programming environment. A much higher degree of interaction is desirable, so that the tools can cooperate and provide functions not feasible with disjoint tools. Such interaction is feasible only if the user's program has a representation accessible through a common interface. One such representation is an attributed parse tree, created by the parser, manipulated by the semantic analyzer, and read by the code generator and debugger [Schatz *et al.* 79, Goos&Wulf 81, Habermann 79, Habermann *et al.* 81]. Another possible representation is the collection of relations generated by the relational monitor, and perhaps used by the compiler and loader for optimization based on performance data [Schwans 82, Segall *et al.* 83, Jones&Schwans 82, Singh 81]. Generalizing this approach, a program ceases to be a collection of typed files

(in essence, a very specialized and restricted database), but rather a general database in which the various utilities insert and extract information useful for the operations they implement [Cattell 80]. This database must incorporate in an integrated fashion both static information, such as the parse tree and the symbol table, and dynamic information, such as that gathered by the monitor. Finally, it would be useful if this database were in some way accessible to the program as it executes. The program could use the information in the database to perform dynamic reconfiguration, error recovery, and load balancing.

A more general extension concerning databases is temporal databases. The application of the relational paradigm to monitoring required a serious examination of the role of time in databases. Recently other researchers have been looking into this area [Bolour *et al.* 82, Bradley 78, Bubenko 77, Anderson 82]. The research reported here has produced significant results, primarily the design, implementation, and formal specification of TQuel. The technology transfer can go both ways, either applying concepts from conventional databases to the relational monitor, or trying to map the ideas behind the relational monitor onto general temporal databases. For example, access nodes are similar to indexed relations, and temporal order is similar to sort order, so techniques involving one concept may also apply to the other concept [Smith&Chang 75, Kim 81, Yao 79, Wong&Youssefi 76].

An even more general and hence, less defined area for future research is the extension of the knowledge base used by the monitor. This knowledge base currently involves the information that is processed, stored, and retrieved, as well as the algorithms performing these manipulations. Extensions to the knowledge base include representing causality in the monitor, and using the knowledge base to make inferences about future events. Extending the TQuel semantics to use temporal logic [Rescher 71] might be a step in this direction.

Clearly, there are many interesting further questions one can ask given the framework developed here. The adventure continues.

References

[Abrams&Treu 77]
M.D. Abrams and S. Treu.
A Methodology for Interactive Computer Service Measurement.
Communications of the ACM 20(12):936-944, December, 1977.

[Aggerwala&Arvind 82]
T. Aggerwala and Arvind.
Data Flow Systems.
IEEE Computer Magazine 15(2):10-13, February, 1982.

[Anderson 82] T.L. Anderson.
Modelling Time at the Conceptual Level.
In *Proceedings of the Second International Conference on Databases.* Jerusalem, June, 1982.

[Andler 79] S.A. Andler.
Predicate Path Expressions: A High-level Synchronization Mechanism.
PhD thesis, Carnegie-Mellon University, Computer Science Department, August, 1979.

[Astrahan&Chamberlin 75]
M. M. Astrahan and D. D. Chamberlin.
Implementation of a structured English query language.
Communications of the ACM 18(10):580-588, October, 1975.

[Ball *et al.* 76] J.E. Ball, J.A. Feldman, J.R. Low, R.F. Rashid, and P.D. Rovner.
RIG, Rochester's Intelligent Gateway: System Overview.
IEEE Transactions on Software Engineering SE-2(4), December, 1976.

[Becerril *et al.* 79] J. Becerril, R. Casajuana, and L. Lorie.
Gsysr: A Relational Database Interface for Graphics.
Springer-Verlag, Florence, Italy, 1979, pages 459-474.

[Berzins&Kapur 77]
V. Berzins and D. Kapur.
Denotational and Axiomatic Definitions for Path Expressions.
Computational Structures Group Memo 153-1, Laboratory for Computer Science, M.I.T., November, 1977.

[Boggs *et al.* 80] D.R. Boggs, J.F. Shoch, E.A. Taft, and R.M. Metcalfe.
Pup: An internetwork architecture.
IEEE Transactions on Communications COM-28(4):612-24, April, 1980.

[Bolour *et al.* 82] A. Bolour, T.L. Anderson, L.J. Debeyser, and H.K.T. Wong.
The Role of Time in Information Processing: A Survey.
SigArt Newsletter 80:28-48, April, 1982.

[Bonner 69] A.J. Bonner.
Using system monitor output to improve performance.
IBM Systems Journal 4:290-298, 1969.

[Boral&DeWitt 81] H. Boral and D.J. DeWitt.
Processor Allocation for Multiprocessor Database Machines.
ACM Transactions on Database Systems 6(2):227-254, June, 1981.

[Boral&DeWitt 82] H. Boral and D.J. DeWitt.
Applying Data Flow Techniques to Data Base Machines.
IEEE Computer Magazine 15(8):57-63, August, 1982.

[Bourne 78] S.R. Bourne.
The UNIX Shell.
The Bell System Technical Journal 57(6, part 2):1971-1990, July-August, 1978.

[Bradley 78] J. Bradley.
Operations Data Bases.
In *Proceedings of the Fourth International Conference on Very Large Data Bases*, pages 164-176. West Berlin, Germany, September, 1978.

[Bubenko 77] J.A. Bubenko.
The Temporal Dimension in Information Modeling.
North-Holland Pub. Co., 1977, .

[Cattell 80] R.G.G. Cattell.
Integrating a Database System and Programming/Information Environment.
In M.L. Brodie and S.N. Zilles (editor), *Proceedings of the Workshop on Data Abstraction, Databases, and Conceptual Modelling*, pages 110-111. ACM, June, 1980.

[Charniak *et al.* 80]
E. Charniak, C.K. Reisbeck, and D.V. McDermott.
Artificial Intelligence Programming.
L. Erlbaum Associates, Hillsdale, NJ, 1980.

[Clark 78] D.D. Clark, K.T. Pogran, and D.P. Reed.
An Introduction to Local Area Networks.
Proceedings of the IEEE 66(11):1497-1516, November, 1978.

[Codd 70] E.F. Codd.
A Relational Model of Data for Large Shared Data Bank.
Communications of the ACM 13(6):377-387, June, 1970.

[Codd 79] E.F. Codd.
Extending the Database Relational Model to Capture More Meaning.
ACM Transactions on Database Systems 4(4):397-434, December, 1979.

[Dahl *et al.* 67] O.J. Dahl *et al.*.
Simula 67, a Common Base Language.
Technical Report, Norwegian Comput. Centre, Forskningveien, Oslo, 1967.

[Dannenberg 81] R.B. Dannenberg.
AMPL: Desing, Implementation, and Evaluation of A Multiprocessing Language.
Technical Report, Carnegie-Mellon University, Computer Science Department, March, 1981.

[Date 76] C.J. Date.
Systems Programming Series: An Introduction to Database Systems.
Addison-Wesley Publishing Company, Reading, MA, 1976.

[Davies *et al.* 79] D.W. Davies, D.L.A. Barber, W.L. Price, and C.M. Solomonides.
Computer Networks and Their Protocols.
John Wiley and Sons, New York, 1979.

[Davis&Keller 82] A.C. Davis and R.M. Keller.
Data Flow Program Graphs.
IEEE Computer Magazine 15(2):26-41, February, 1982.

[Dennis 80] J.B. Dennis.
Data Flow SuperComputers.
IEEE Computer Magazine 13(11):48-56, November, 1980.

[Enslow 78] P.H. Enslow, Jr.
What is a distributed processing system ?
IEEE Computer Magazine 11(1), January, 1978.

[Fabry 74] R.S. Fabry.
Capability-based addressing.
Communications of the ACM 17(7):403-12, July, 1974.

[Feldman&Rovner 69]
J. Feldman and P. Rovner.
An Algol-Based Associative Language.
Communications of the ACM 12(8):439-449, August, 1969.

[Foderaro 80] J.K. Foderaro.
FRANZLisp Manual
Opus 33b edition, UC Berkeley, 1980.

[Forgy 79] C.L. Forgy.
On the Efficient Implementation of Production Systems.
PhD thesis, Carnegie-Mellon University, Computer Science Department, February, 1979.

[Fuller *et al.* 78] S. Fuller, J. Ousterhout, L. Raskin, P. Rubinfeld, P.Sindhu and R. Swan.
Multi-microprocessors: An overview and working example.
Proceedings of the IEEE 66(2):216-28, February, 1978.

[Gehringer&Chansler 82]
E.F. Gehringer and R.J. Chansler, Jr.
StarOS User and System Structure Manual.
Technical Report, Carnegie-Mellon University, Computer Science Department, July, 1982.

[Gertner 80] I. Gertner.
Performance Evaluation of Communicating Processes.
PhD thesis, Computer Science Department, University of Rochester, May, 1980.
Available as TR 74.

[Goos&Wulf 81] G. Goos and W.A. Wulf (eds).
Diana Reference Manual.
Technical Report CMU-CS-81-101, Carnegie-Mellon University, Computer Science Department, March, 1981.
Descriptive Intermediate Attributed Notation for Ada.

[Habermann 75] A.N. Habermann.
Path Expressions.
Technical Report, Carnegie-Mellon University, Computer Science Department, June, 1975.

[Habermann 79] A.N. Habermann.
An Overview of the Gandalf Project.
Carnegie-Mellon University, Computer Science Research Review , 1978-79.

[Habermann *et al.* 81]
N. Habermann, D. Perry, P. Feiler, R. Medina-Mora, D. Notkin, G. Kaiser, and B. Denny.
A Compendium of Gandalf Documentation.
Technical Report, Carnegie-Mellon University, Computer Science Department, April, 1981.

[Held *et al.* 75] G.D. Held, M. Stonebraker, and E. Wong.
INGRES--A relational data base management system.
Proceedings of the 1975 National Computer Conference 44:409-416, 1975.

[Herot 80] C.F. Herot.
Spacial Management of Data.
ACM Transactions on Database Systems 5(4):493-513, December, 1980.

[Highnam 81] P.T. Highnam.
Medic: A Resident Monitor for Medusa.
Multiprocessor Performance Evaluation Group Internal Memo, Carnegie-Mellon University, Computer Science Department, September, 1981.

[Highnam&Snodgrass 81]
P.T. Highnam and R.T. Snodgrass.
The Cm/Simon Protocol.*
Multiprocessor Performance Evaluation Group Internal Report, Carnegie-Mellon University, Computer Science Department, 1981.

[Hoare 74] C.A.R. Hoare.
Monitors: An Operating System Structuring Concept.
Communications of the ACM 17(10), October, 1974.

[Ichbiah *et al.* 79] J.D. Ichbiah *et al.*.
Rationale for the Design of the ADA Programming Language.
SIGPLAN Notices 14(6), June, 1979.

[Ingalls 78] D. Ingalls.
The Smalltalk-76 Programming System: Design and Implementation.
In *Proceedings of the Fifth Principles of Programming Languages*, pages 9-16. ACM, January, 1978.

[Johnson 75] S.C. Johnson.
Yacc: Yet Another Compiler Compiler.
Computing Science Technical Report 32, Bell Laboratories, Murray Hill, NJ, 1975.

[Jones 77] A.K. Jones.
The Narrowing Gap Between Language Systems and Operating Systems.
In *Proceedings of the IFIP Conference*. 1977.

[Jones *et al.* 79] A.K. Jones, R.J. Chansler, Jr., I. Durham, J. Mohan, K. Schwans, and S. Vegdahl.
StarOS, a Multiprocessor Operating System.
In *Proceedings of the Seventh Symposium on Operating System Principles*, pages 117-127. ACM/SIGOPS, Asimolar Conference Grounds, Pacific Grove, CA, December, 1979.

[Jones *et al.* 78] A.K. Jones, R.J. Chansler, Jr., I. Durham, P. Feiler, D. Scelza, K. Schwans, and S.R. Vegdahl.
Programming issues raised by a multiprocessor.
Proceedings of the IEEE 66(2):229-37, February, 1978.

[Jones&Gehringer 80]
A.K. Jones and Ed Gehringer, eds.
The Cm Multiprocessor Project: A Research Review.*
Technical Report CMU-CS-80-131, Carnegie-Mellon University, Computer Science Department, July, 1980.

[Jones&Schwans 79]
A.K. Jones and K. Schwans.
TASK Forces: Distributed Software for Solving Problems of Substantial Size.
In *Proceedings of the 4th International Conference on Software Engineering*. ACM, IEEE, Munich, W. Germany, September, 1979.

[Jones&Schwans 80]
A.K. Jones and K. Schwans.
The TASK Specification Language.
StarOS Group Internal Report, Carnegie-Mellon University, Computer Science Department, 1980.

[Jones&Schwans 82]
A.K. Jones and K. Schwans.
Specifying Software Configuraticns for Distributed Computation.
1982.
To be published.

[Kahn *et al.* 81] K.C. Kahn, W.M. Corwin, T.D. Dennis, H. D'Hooge, D.E. Hubka, L.A. Hutchins, J.T. Montague, F.J. Pollack, and M.R. Gifkins.
iMAX: A Multiprocessor Operating System for an Object-Oriented Computer.
In *Proceedings of the Eighth Symposium on Operating Systems Principles*, pages 127-136. ACM/SIGOPS, December, 1981.

[Kahn&Gorry 75] K. Kahn and G.A. Gorry.
Mechanizing Temporal Knowledge.
Artificial Intelligence 9:87-108, September, 1975.

[Kernighan&Ritchie 78]
B.W. Kernighan and D.M. Ritchie.
Prentice-Hall Software Series: *The C Programming Language.*
Prentice-Hall, Englewood Cliffs, NJ, 1978.

[Kim 81] W. Kim.
On Optimizing SQL-like nested queries.
Technical Report RJ 3063, IBM, San Jose, CA, 1981.

[Kleene 38] S.C. Kleene.
On a Notation for Ordinal Numbers.
Journal of Symbolic Logic 3:150-155, 1938.

[Lamport 78] L. Lamport.
Time, Clocks, and the Ordering of Events in a Distributed System.
Communications of the ACM 21(7):558-565, July, 1978.

[Lauer&Campbell 75]
P.E. Lauer and R.H. Campbell.
Formal Semantics of a Class of High-Level Primitives for Coordinating Concurrent Processes.
Acta Informatica 5:297-332, 1975.

[Leverett *et al.* 80] B. Leverett, R.G.G. Cattell, S.O. Hobbs, J.M. Newcomer, A.H. Reiner, B.R. Schatz, and W.A. Wulf.
An Overview of the Production Quality Compiler-Compiler Project.
IEEE Computer Magazine 13(8):38-49, August, 1980.

[Lipski 79] W. Lipski, Jr.
On databases with incomplete information.
1979.
Unpublished technical report.

[Liskov 81] B. Liskov, ed.
Report on the Workshop on Fundamental Issues in Distributed Computing.
Op Sys Review 15(3), July, 1981.

[McDermott 80] D. McDermott.
Spacial Inferences with Ground, Metric Formulas on Simple Objects.
Research Report 173, Yale University, Department of Computer Science, January, 1980.

[Metcalfe&Boggs 75]
R.M. Metcalfe and D.R. Boggs.
Ethernet: Distributed Packet Switching for Local Computer Networks.
Technical Report CSL-75-7, Xerox Palo Alto Research Center, Palo Alto, CA., November, 1975.

[Model 79] M. Model.
Monitoring System Behavior in a Complex Computational Environment.
Technical Report CSL-79-1, Xerox PARC, January, 1979.

[Morgan *et al.* 75] David E. Morgan, Walter Banks, Dale P. Goodspeed, and Richard Kolanko.
A Computer Network Monitoring System.
IEEE Transactions on Software Engineering SE-1(3), September, 1975.

[Nelson 81] B.J. Nelson.
Remote Procedure Call.
PhD thesis, Carnegie-Mellon University, Computer Science Department, May, 1981.
also published as Xerox PARC Technical Report CSL-81-9.

[Nutt 79] G.J. Nutt.
A Survey of Remote Monitors.
Special Publication 500-42, National Bureau of Standards, January, 1979.

[Ousterhout *et al.* 80]
J. K. Ousterhout, D. A. Scelza, and P. S. Sindhu.
Medusa: an experiment in distributed operating system structure.
Communications of the ACM 23(2), February, 1980.

[Plattner&Nievergelt 81]
B. Plattner and J. Nievergelt.
Monitoring Program Execution: A Survey.
IEEE Computer Magazine 16(11):76-93, November, 1981.

[Rashid 80a] R.F. Rashid.
A Network Operating System for a Distributed Sensor Network.
Internal memo, Carnegie-Mellon University, Computer Science Department, April, 1980.

[Rashid 80b] R.F. Rashid.
An Inter-Process Communication Facility for UNIX.
Technical Report CMU-CS-80-124, Carnegie-Mellon University, Computer Science Department, March, 1980.

[Reiter 78] R. Reiter.
On closed world data bases.
Plenum Press, New York, 1978, .

[Rescher 68] N.C. Rescher.
Many-valued Logic.
McGraw-Hill, New York, 1968.

[Rescher 71] N.C. Rescher and A. Urquhart.
Temporal Logic.
Springer-Verlag, New York, 1971.

[Ritchie&Thompson 74]
D.M. Ritchie and K. Thompson.
The UNIX time-sharing system.
Communications of the ACM 17(7):365-75, July, 1974.

[Rosen 81] E. Rosen.
Vulnerabilities of Network Control Protocols: An Example.
ACM SIGSOFT Soft Eng Notes , January, 1981.

[Rosenberg 83] J. Rosenberg.
Generating Efficient Code for Generics.
PhD thesis, Carnegie-Mellon University, Computer Science Department, May, 1983.
To be completed.

[Saunders 79] S.E. Saunders.
Compiling Customized Executable Representations and Interpreters.
PhD thesis, Carnegie-Mellon University, Computer Science Department, June, 1979.
Available as TR CS-79-127.

[Schatz *et al.* 79] B.R. Schatz, B.W. Leverett, J.M. Newcomer, A.H. Reiner, and W.A. Wulf.
TCOLAda: An Intermediate Representation for the DOD Standard Programming Language.
Technical Report, Carnegie-Mellon University, Computer Science Department, March, 1979.

[Schwans 82] K. Schwans.
Configuring software for multiple processor systems.
PhD thesis, Carnegie-Mellon University, Computer Science Department, September, 1982.

[Segall *et al.* 83] Z. Segall, A. Singh, R.T. Snodgrass, A.K. Jones, and D.P. Siewiorek.
An Integrated Instrumentation Environment for Multiprocessors.
IEEE Computer Magazine , 1983.
to be published.

[Shaw 80] A.C. Shaw.
Software Specification Languages Based on Regular Expressions.
Springer-Verlag, 1980, .

[Shaw 81] Mary Shaw, ed.
ALPHARD: Form and Content.
Springer-Verlag, New York, 1981.

[Shoch 79] John Shoch.
EFTP: A Pup-based Ether file transfer protocol.
1979.
Unpublished specification.

[Shoch 81] J. Shoch.
What's Different About 'Distributed Computing'?
1981.
in [Liskov 81].

[Shrobe 79] H. Shrobe.
Dependency Directed Reasoning for Complex Program Understanding.
PhD thesis, MIT, 1979.
published as MIT Technical Report AI-TR-503.

[Singh 81] A. Singh.
Pegasus-A Workload Generator for Multiprocessors.
Master's thesis, Carnegie-Mellon University, Department of Electrical Engineering, 1981.

[Smith&Chang 75]
J.M. Smith and P.Y.-J. Chang.
Optimizing the Performance of a Relational Algebra Database Interface.
Communications of the ACM 18(10):568-579, October, 1975.

[Snodgrass 82] R.T. Snodgrass.
Description File Specifications.
Multiprocessor Performance Evaluation Group Internal Report, Carnegie-Mellon University, Computer Science Department, August, 1982.

[Snodgrass 83] R.T. Snodgrass.
An Object-Oriented Command Language.
IEEE Transactions on Software Engineering 9(1), January, 1983.

[Stonebraker *et al.* 76]
M. Stonebraker, E. Wong, P. Kreps, and G. Held.
The Design and Implementation of INGRES.
ACM Transactions on Database Systems 1(3):189-222, September, 1976.

[Swan *et al.* 77] Richard J. Swan, Samuel H. Fuller and D. P. Siewiorek.
Cm*: A modular, multi-microprocessor.
In *Proceedings of the National Computer Conference*, pages 637-44.
AFIPS, 1977.

[Tobagi *et al.* 76] F.A. Tobagi, S.E. Lieberson, and L. Kleinrock.
On measurement facilities in packet radio systems.
In *Proceedings of the National Computer Conference*, pages 589-596.
AFIPS, 1976.

[Ullman 82] J.D. Ullman.
Principles of Database Systems, Second Edition.
Computer Science Press, Potomac, Maryland, 1982.

[Vassiliou 79] Y. Vassiliou.
Null values in database management--a denotational semantics approach.
In *Proceedings of the International Symposium on Management of Data*,
pages 162-169. ACM/SIGMOD, 1979.

[Vegdahl 82] S. Vegdahl.
private communication.

[Williams 74] R. Williams.
On the Application of Relational Data Structures in Computer Graphics.
In *Information Processing 74*, pages 722-726. IFIP, 1974.

[Winston&Horn 81]
P.K. Winston and B.K.P. Horn.
Lisp.
Addison-Wesley, Hillsdale, NJ, 1981.

[Wirth 71] N. Wirth.
The Programming Language Pascal.
Acta Informatica 1(1), 1971.

[Wong&Youssefi 76]
E. Wong and K. Youssefi.
Decomposition--A Strategy of Query Processing.
ACM Transactions on Database Systems 1(3):223-241, September, 1976.

[Wulf *et al.* 75a] W.A. Wulf, R.Levin, and C.Pierson.
Overview of the Hydra Operating System.
In *Proceedings of the Fifth Symposium on Operating System Principles.* ACM/SigOps, November, 1975.

[Wulf *et al.* 75b] W.A. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke.
Bliss-11 Manual.
Technical Report, Carnegie-Mellon University, Computer Science Department, 1975.

[Wulf *et al.* 75c] W.A. Wulf, R.K. Johnsson, C.B. Weinstock, S.O. Hobbs, and C.M. Geschke.
Elsevier Computer Science Library: The Design of an Optimizing Compiler.
Elsevier, New York, 1975.

[Wulf *et al.* 81] W.A. Wulf, R. Levin, and S.P. Harbison.
HYDRA/C.mmp: An Experimental Computer System.
McGraw-Hill, 1981.

[Yao 79] S.B. Yao.
Optimization of Query Evaluation Algorithms.
ACM Transactions on Database Systems 4(2):133-155, June, 1979.

Appendix A
BNF of the TQuel Retrieve Statement

This appendix lists the syntax for the TQuel retrieve statement. Since TQuel is a strict superset of Quel, all legal Quel retrieve statements are also legal TQuel retrieve statements. The following non-terminals are not included in the syntax description because they are identical to their Quel counterparts.

<bool expression>	returns a value of type boolean
<expression>	returns a value of type integer, string, floating point, or temporal
<integer>	an integer constant
<name>	the name of a domain
<relation>	a relation name
<string>	a string constant
<tuple variable>	the name of a tuple variable

Also not shown are the additional temporal functions and predefined relations found in TQuel.

The syntax is given in standard BNF, with non-terminals in "<>". The empty string is denoted by "ε". Terminals identical to the meta-character "|" are enclosed in double quotation marks.

```
<TQuel retrieve>     ::= <retrieve head> <retrieve tail>

<retrieve head>      ::= retrieve <into> <target list>

<into>     ::= ε | unique | <relation> | into <relation>

<target list>        ::= ε | ( <tuple variable> . all ) | ( <t-list> )

<t-list>    ::= <t-elem> | <t-list> , <t-elem>
```

```
<t-elem>    ::= <name> <is> <expression>

<is>        ::= is | =

<retrieve tail>     ::= <selection> <temporal delimiter>

<selection> ::= <where clause> <temporal selection>

<where clause>      ::= ε | where <bool expression>

<temporal selection> := ε | when <tbool-exp>

<tbool-exp> ::= <t-exp>
            | ( <tbool-exp> )
            | <tbool-exp> and <tbool-exp>
            | <tbool-exp> or <tbool-exp>
            | not <tbool-exp>

<t-exp>     ::= <element>
            | <t-exp> . time
            | <t-exp> . start
            | <t-exp> . stop
            | <t-exp> ; <t-exp>
            | <t-exp> , <t-exp>
            | <t-exp> "|" <t-exp>
            | ( <t-exp> )

<element>   ::= <tuple variable>
            | <string>
            | <integer>

<temporal delimiter> ::= <period delimiter> | <at clause>

<period delimiter>  ::= <start clause> <stop clause>

<start clause>      ::= ε | start <event expression>

<stop clause>       ::= ε | stop <event expression>

<at clause> ::= ε | at <event expression>

<event expression>  ::= <element>
            | <event expression> . time
            | <event expression> . start
            | <event expression> . stop
            | <event expression> ; <event expression>
            | <event expression> , <event expression>
            | ( <event expression> )
```

Appendix B
Proof of the Conversion Theorem

Before presenting the proof of the Conversion Theorem used in chapter 4, it is useful to repeat the syntax of temporal expressions and to list the productions. The syntax of the temporal expressions is as follows:

```
<t-exp>    ::= <tuple variable>
             | <t-exp> . time
             | <t-exp> . start
             | <t-exp> . stop
             | <t-exp> ; <t-exp>
             | <t-exp> "|" <t-exp>
             | <t-exp> , <t-exp>
             | ( <t-exp> )
```

Seventeen productions are used to specify the semantics of the above syntax[27]:

(1) $a\,.\,\text{start} \quad \Rightarrow a$

(2) $(\alpha \mid \beta)\,.\,\text{start} \quad \Rightarrow (\alpha\,.\,\text{start} \mid \beta\,.\,\text{start})$

(3) $(\alpha\,;\,\beta)\,.\,\text{start} \quad \Rightarrow \alpha\,.\,\text{start}$

(4o) $(\alpha\,,\,\beta)\,.\,\text{start} \quad \Rightarrow (\alpha\,.\,\text{start}\,;\,\beta\,.\,\text{start}) \mid (\beta\,.\,\text{start};\,\alpha\,.\,\text{start})$

(4c) $(\alpha\,,\,\beta)\,.\,\text{start} \quad \Rightarrow (\alpha\,.\,\text{start} \mid \beta\,.\,\text{start})$

(5) $a\,.\,\text{stop} \quad \Rightarrow a$

(6) $(\alpha \mid \beta)\,.\,\text{stop} \quad \Rightarrow (\alpha \mid \beta)$

---

[27] Both the overlap and coverage interpretations are accommodated here. The productions associated with the overlap interpretation are denoted by (o), and with the coverage interpretation by (c).

(7)	$(\alpha ; \beta)$. stop	$\Rightarrow \beta$. stop
(8o)	(α , β) . stop	$\Rightarrow (\alpha$. stop $\mid \beta$. stop)
(8c)	(α , β) . stop	$\Rightarrow (\alpha$. stop ; β . stop) $\mid (\beta$. stop ; α . stop)
(9)	(α)	$\Rightarrow \alpha$
(10o)	a , $(\beta ; \gamma)$	$\Rightarrow$ a
(10c)	a , $(\beta ; \gamma)$	$\Rightarrow$ ((a , β) ; γ) $\mid$ (β ; (a , γ)) $\mid$ (β . stop ; a ; γ . start)
(11)	$\alpha , (\beta \mid \gamma)$	$\Rightarrow (\alpha , \beta) \mid (\alpha , \gamma)$
(12)	$(\alpha \mid \beta) , \gamma$	$\Rightarrow (\alpha , \gamma) \mid (\beta , \gamma)$
(13o)	(a ; α) , b	$\Rightarrow$ (a ; b ; α) $\mid$ (a ; (α , b))
(13c)	(a ; α) , b	$\Rightarrow$ (b ; a ; α) $\mid$ (a ; α ; b) $\mid$ (a ; (b , α))

(14o) $(a_1 ; \ldots ; a_j , (b_1 ; \ldots ; b_k) \Rightarrow (a_1 ; b_1 \mid b_1 ; a_1) ; (a_j \mid b_k)$

(14c) $(a_1 ; \ldots ; a_j) , (b_1 ; \ldots ; b_k) \Rightarrow (a_1 \mid b_1) ; (a_j ; b_k \mid b_k ; a_j)$

(15)	$\alpha ; (\beta \mid \gamma)$	$\Rightarrow (\alpha ; \beta) \mid (\alpha ; \gamma)$
(16)	$(\beta \mid \gamma) ; \alpha$	$\Rightarrow (\beta ; \alpha) \mid (\gamma ; \alpha))$
(17c)	a , b	$\Rightarrow$ (a ; b $\mid$ b ; a)

Production (17c) says that, under the coverage interpretation, when two events occur in parallel, one of them will come first, and the other second, with the result being the period between them. (17c) is not valid under the parallel interpretation of the parallel operator, since in that case the events must have occurred at exactly the same moment.

Theorem 1: The productions eliminate all . start terms.

Proof: By induction on the number of terminals in the original expression α.

Basis: k = 1. α is either (a) the terminal a, (b) a . start, (c) a . stop, or (d) (α). In cases (a) and (c), there are no . start terms in α. (9) will eliminate nested parentheses, taking care of case (d). (1) will eliminate the . start term in case (b).

Inductive step: Assume the theorem is true for all expressions containing less than k terminals. Assume α contains k terminals. Then α must be of the form (after multiple parentheses have been eliminated by (9)):

(a) $(\beta \mid \gamma)$	(d) $(\beta \mid \gamma)$. start	(g) $(\beta \mid \gamma)$. stop
(b) $(\beta ; \gamma)$	(e) $(\beta ; \gamma)$. start	(h) $(\beta ; \gamma)$. stop
(c) (β , γ)	(f) (β , γ) . start	(i) (β , γ) . stop

Since β and γ each contain fewer than k terminals, they can be transformed into β' and γ', not containing . start terms. Substituting β' and γ' for β and γ in the above expressions, (a) - (c) and (g) - (i) do not contain any . start terms. Applying (2) to (d), we get

$$(\beta \mid \gamma) \text{ . start} \Rightarrow (\beta \text{ . start} \mid \gamma \text{ . start})$$

Since β . start contains fewer than k terminals, it can be transformed into a β without . start terms, and similarly for γ . start, yielding an expression containing no . start terms. The same holds true for (e) and (f) by using (3) and (4) respectively.

□

Corollary: The productions eliminate all . stop terms.

Lemma 2: Any regular expression (involving | and ;) can be converted into standard form.

Proof: By induction on the number of terminals in the original expression α.

Basis: k = 1. Already in standard form.

Inductive step: Assume lemma is true for all expressions containing less than k terminals. Assume α contains k terminals. α is either $(\beta ; \gamma)$ or $(\beta \mid \gamma)$, for some β and γ. β and γ can be converted into standard form. In the first case, by (15) and (16),

$$\begin{aligned}(\beta ; \gamma) \equiv (\beta_1 \mid \beta_2 \mid \cdots \mid \beta_m ; (\gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_n) \\ \Rightarrow (\beta ; \gamma_1) \mid (\beta ; \gamma_2) \mid \cdots \mid (\beta ; \gamma_n) \\ \Rightarrow (\beta_1 ; \gamma_1) \mid \cdots \mid (\beta_m ; \gamma_n)\end{aligned}$$

which is in standard form. In the second case, α is already in standard form, due to the associativity of the selection operator ("|").

□

Note that converting into standard form usually increases the number of terminals (though not the number of *unique* terminals).

Lemma 3: An expression α of the form (β , γ), where β and γ are in standard form, can be transformed by the productions into an expression not containing the parallel operator.

Proof: Induction on the number of terminals in α.

Basis: k = 2. α = (a , b), where a and b are terminals. Apply (17c).

Inductive step: Assume lemma is true for all expressions involving less than k terminals. Assume α contains k terminals. Since β is in standard form, it must be either (a) a single terminal b, (b) a single sequence of terminals (b_1 ; . . . ; b_j), or (c) an alternation of sequences $\beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$.

(a) β is a single terminal. γ must be either (d) a single terminal, (e) a single sequence of terminals, or (f) an alternation of sequences. (17c) eliminates the parallel operator from α in case (d). In (e) and (f), (10) and (11) transform α into an expression containing subexpressions which contain the parallel operator yet which are in standard form and involve less than k terminals, allowing the parallel operator to be eliminated.

(b) β is a sequence of terminals. γ must be either (g) a single terminal, (h) a sequence of terminals, or (i) a selection of sequences, which may be handled by applying (13), (14), and (11), respectively.

(c) Use (12).

□

Conversion Theorem: The productions convert any temporal expression into a standard expression.

Proof: By induction on the number of terminals. Let α be the original expression where all . start and . stop terms have been eliminated using Theorem 1 and its corollary.

Basis: k = 1. α cannot contain a parallel operator.

Inductive step: Assume the theorem is true for all expressions containing less than k terminals. Assume α contains k terminals. After eliminating multiple parentheses with (9), α is of the form (a) $(\beta \mid \gamma)$, (b) $(\beta ; \gamma)$, (c) (β , γ) where β and γ have had their parallel operators eliminated. In cases (a) and (b), α no longer contains a parallel operator. For case (c), use Lemma 2 to transform β and γ into standard form, then use Lemma 3 to eliminate the parallel operator.

At this point, the expression contains only sequence and selection operators, and can be converted into standard form by Lemma 2.

□

Note that the proof applies regardless of the interpretation placed on the parallel operator. Also note that all the productions ((1) - (17)) were used in the proof. Thus, the set of productions has been shown to be sufficient, and appears to be necessary, to convert a temporal expression into a standard (regular) expression.

Appendix C
Operator Nodes

This appendix consists of descriptions of the operator nodes. The description of each node consists of its signature (the node name followed in turn by the instantiation parameters in parentheses) followed by an explanation and example of the computation performed by the node. Unless otherwise noted, an output tuple is generated for each input tuple. Except for the Cartesian and join nodes, all operator nodes are unary.

Domains are specified by the integer index of the domain (starting with 0), with -1 indicating no domain is applicable. The time domain(s) implicit in TQuel is explicit in the update network; indeed, most operator nodes take as arguments the indices of the start and stop domains. Examples use the following primitive period relation

```
A (Process, B, C, D)
```

which has 6 domains: (0) start-time, (1) Process, (2) B, (3) C, (4) D, and (5) stop-time. See the explanation below of this unusual numbering of domains. The example code is quite similar to that produced by the monitor, and has intentionally been left unoptimized for easier understanding.

EventToPeriod (ArgDomain StopDomain); unary

Events associated with a traced primitive relation are automatically converted to periods. The ArgDomain indicates either the sensor or the object domain (depending on whether the event is sensor or object traced); the StopDomain specifies where the computed stop time is to be placed. In the following example, the newly computed stop domain is appended to the tuple, and hence must be domain 5.

Example:

```
retrieve L (A.all)

⇒

(create A access-1)
(create eventtoperiod eventtoperiod-1 (1 5))
(link access-1 eventtoperiod-1 left)
```

Projection (Result(s)); unary

The specified domains are selected from the incoming tuple and concatenated to form the output tuple.

Example:

```
retrieve M (B = L.B)

⇒

(create projection projection-1 ((0 5 2)))
(link eventtoperiod-1 projection-1 left)
```

ApplyOp (ResultDomain Function Argument(s)); unary

This node computes a new domain whose value is the result of performing the specified function on one or more domains of the input tuple. The function may be any Lisp function; special semantics are associated with arithmetic operations on temporal domains (see section 8.6.1).

Example:

```
retrieve N (B = L.B + L.C)

⇒

(create applyop applyop-1 (6 (lambda ($2 $3) (+ $2 $3)) (2 3)))
(link eventtoperiod-1 applyop-1 left)
(create projection projection-2 ((0 4 5)))
(link applyop-1 projection-2 left)
```

Display (Heading DomainName(s)); unary

The Display node maintains a display of the argument relation on the screen.

Example:

```
display N

⇒

(create display display-1 (N (Start Stop B)))
(link projection-2 display-1 left)
```

RelationStore (RelationName DomainType(s)); unary

The incoming tuples are stored in a standard Ingres relation.

Example:

```
store N in NRelation

⇒

(create relationstore relationstore-1
```

```
                    (NRelation (time time integer)))
(link projection-2 display-1 left)
```

StoreFile (FileName DomainType(s)); unary

The incoming tuples are stored in the specified file.

Example:

```
store N in file NFile

⟹

(create storefile storefile-1 (NFile (time time integer)))
(link projection-2 storefile-1 left)
```

Cartesian (StartDomain StopDomain); binary

The output tuple contains all of the domains of the two inputs. Each time a tuple enters the left input, it is concatenated with all of the tuples that have entered the right input, with the resulting tuples put on the output arc. An analogous process occurs each time a tuple enters the right input.

Example:

```
retrieve P (B1 = M.B, B2 = N.B)
start M.start
stop N.stop

⟹

(create cartesian cartesian-1)
(join projection-1 cartesian-1 left)
(join projection-2 cartesian-1 right)
(create projection-3 projection ((0 1 2 5)))
(link cartesian-1 projection-3 left)
```

Selection (Predicate Argument(s)); unary

Tuples satisfying the predicate are placed on the output arc.

Example:

```
retrieve Q (L.all)
where L.B = 1

⟹

(create selection selection-1 ((lambda ($2) (eq $2 1)) (2)))
(link eventtoperiod-1 selection-1 left)
```

Join (StartDomain StopDomain Predicate Argument(s)); binary

The output contains all of the domains of the two inputs. Each time a tuple enters the first

input, it is concatenated with all of the tuples that have entered the second input. The predicate is then applied to the resulting tuple. If the predicate is satisfied, the concatenated tuple is placed on the output arc. An analogous process occurs each time a tuple enters the second input. This operator node is equivalent to a Cartesian operator node connected to a Selection operator node.

Example:

```
retrieve R (N.all)
where N.B = P.B
start N.start
stop N.stop

⟹

(create join join-1 ((lambda ($2 $5) (equal $2 $5)) (2 5)))
(link projection-2 join-1 left)
(link projection-3 join-1 right)
(create projection-4 ((0 1 2)))
(link join-1 projection-4 left)
```

AggrOp (Function Where Class Argument Result StartDomain StopDomain); unary

The function specifies the aggregate operator (e.g., count, countc, etc.). If the value of the where domain is -1, then the tuple is ignored. The value of the class domain is used to partition the tuples; again, a value of -1 implies that all tuples are of the same class. The aggregate is applied to the argument domain, with the result of the operation stored in the result domain. The where argument is derived from the where-clause and the class argument from the by-clause.

Example:

```
retrieve S (B = AvgC(L.B by L.C where L.D = 1))

⟹

(create applyop applyop-1 (6 (lambda ($4) (equal $4 1)) (4)))
(link eventtoperiod-1 applyop-1 left)
(create aggrop aggrop-1 (avgc 6 3 2 7))
(link applyop-1 aggrop-1 left)
(create projection-5 projection ((0 5 7)))
(link aggrop-1 projection-5 left)
```

SwitchDisposition; unary

When an event is derived from a period, the *disposition* of the tuples must be changed from event to period. Similarly, when deriving an event from several periods, disposition must also be changed.

Example:

```
retrieve T (L.all)
at L.start
```

⟹

```
(create projection projection-6 ((0 1 2 3)))
(link eventtoperiod-1 projection-6 left)
(create switchdisposition switchdisposition-1)
(link projection-6 switchdisposition-1 left)
```

Order; unary

This node stores incoming events internally until a checkpoint tuple arrives. At that point, the events which have times prior to the time the checkpoint was generated are output in temporal order.

PeriodToEvent(StartDomain StopDomain); unary

For each incoming period tuple, a start period and a stop period tuple are output.

Appendix D
StarMon

This appendix describes in some detail the various algorithms and data structures employed in StarMon, the resident monitor for the StarOS operating system. A brief summary of this material may be found in section 8.3.

D.1. The StarOS Task Force

StarMon, being part of StarOS, is a module in the StarOS task force. Other modules in this task force include the loader, process creator, file system, user interface, and debugger. A task force is implemented as an object of type *catalogue* (see Figure D-1), representing a mapping between module names (character strings) and modules. One of the names in the catalogue is 'Sensor-Description', which is mapped not into a module, but into the remote description of the sensors located in the task force (the Sensor Description Object).

Every process running under StarOS is given access to another catalogue, called the *StarOSLibrary* when the process is created. One of the objects residing in the StarOSLibrary is the *monitor object* (see Figure D-2). Contained in this object is everything needed by an application process (or StarOS process) to communicate with StarMon.

D.2. The Monitor Object

The monitor object is a small object containing two words of data and several capabilities (a capability is a protected pointer [Fabry 74]). In the figures of objects in this appendix, the data and capability portions are separated by a heavy line.

Newly created receptacles are automatically sent to the *receptacle mailbox* so that StarMon may later manipulate them. The *declare* and *delete* mailboxes are interfaces between the garbage collector and StarMon (see section 5.5.1). Event records are placed by each sensor into one of the *pipe*s (so-called because they function much like Unix pipes [Ritchie&Thompson 74]) referenced in the monitor object.

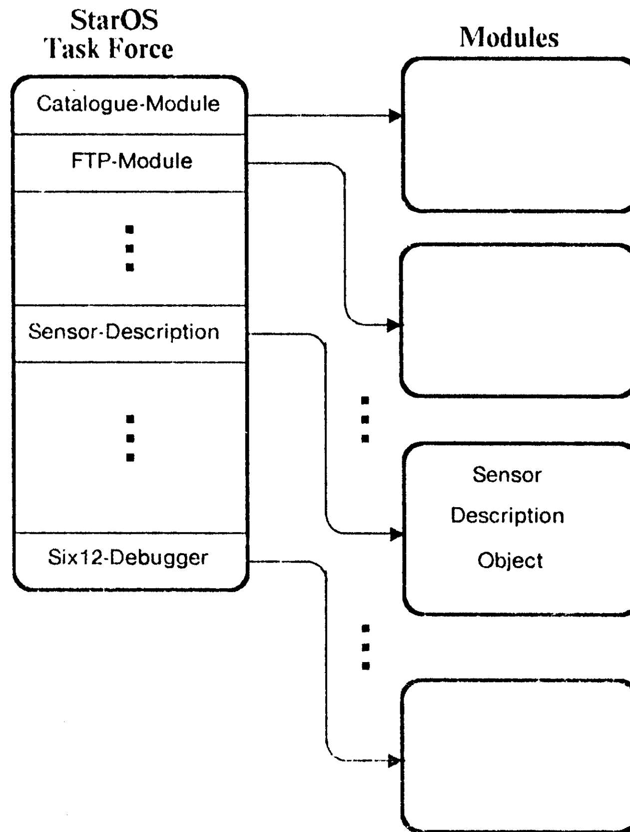

Figure D-1: Representation of the StarOS taskforce

D.3. Pipes

The structure of a pipe is shown in Figure D-3 and that of an event record in Figure D-4. Event records are allocated from the event record buffer in fixed length blocks (the length is specified in the *RecordSize* field). Synchronization is handled by the *Stack* and *Queue* objects referenced in the pipe. These objects, supported by StarOS in microcode, allow single words (2 bytes) to be pushed or popped atomically. The current implementation uses 4 pipes, each containing 100 blocks of 40 bytes, totaling approximately 16 Kbytes, representing a memory overhead of approximately 2%.

The use of the stack and the queue allow multiple processes to add and remove event records. Block addresses, which indicate the starting offset of the block in the event record

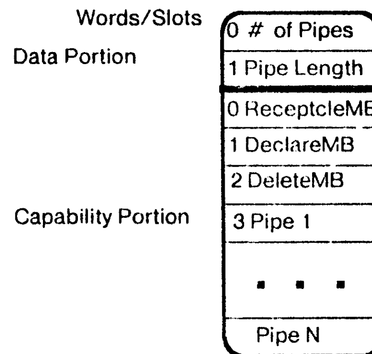

Figure D-2: The Monitor Object

buffer of the pipe, are stored in these objects. The presence of a block address on the stack indicates that the block contains no useful information. A block address on the queue indicates the block contains a valid event record. An address on neither indicates the event record is being entered or removed. When the pipe is first initialized, the queue contains no entries, and the stack contains the addresses of all the blocks in the event record buffer. To add an event record to a pipe, the address of an event record block is first popped off the stack. The event record, in a suitable format, is then written into the buffer, and the address is pushed onto the back of the queue. This set of actions is reversed to remove an event record: the process pops an address from the front of the queue, copies the event record into an internal buffer, and the pushes the address of the block back onto the stack. A stack is used because pushes are slightly more efficient (four instructions verses six for a queue); a queue must be used for the addresses of the filled blocks in order to implement a first-in-first-out strategy.

Since the operating system guarantees indivisibility of the push and pop operations, the store and remove event record operations described above are also indivisible with respect to an individual event buffer. If an error occurs when a block address in on neither the stack nor the queue, then in worst case the address is never pushed, and the use of the block is effectively lost (this condition is difficult to detect, because the process may be waiting to be rescheduled to finish processing the event). However, the information contained in the other blocks is not corrupted in any way as the result of a block address being lost. Unfortunately, since the buffer is shared by all processes having capabilities for the pipe, valid event records can be corrupted if address not within the block indicated by the popped address are written. Since the code for the store and remove operations is small, and only needs to be written once, the probability of an error occurring due to incorrect logic is rather small.

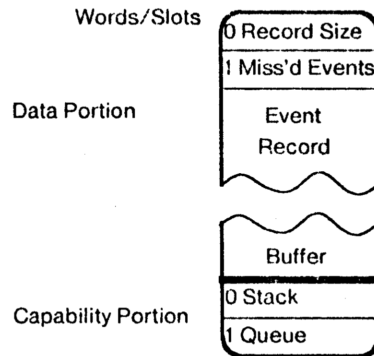

Figure D-3: The Structure of a StarOS Pipe

D.4. Receptacles and Sensors

A StarOS receptacle (see Figure D-5) is an object containing a few words of data and three capabilities. The *Name* field contains the remote name for the object containing the receptacle, and the *Target* field contains the internal name (i.e., the capability) for the same object. The *DeclareMB* field contains a capability to another StarOS object, a mailbox, where the sensor can send capabilities to be placed in StarMon's internal tables. The *DeclareMB* field of the Monitor Object references the same mailbox. Capabilities are sent to MonProc because the event records, sent to the StarMon Accountant, may not contain capabilities, primarily for efficiency reasons. All three fields are not functionally necessary, and are present simply to make event record storage a faster operation. Each receptacle also contains a capability in the *Pipe* field for one of the pipes in the monitor object. The *Lock* field is used for arbitrating modifications to this receptacle by multiple MonProc processes.

Space in each object to be monitored must be provided; the StarOS implementation requires one data word and one capability (a total of six bytes). The data word simply indicates whether the capability is present and is included for efficiency, since checking for a null receptacle capability is a relatively expensive operation (16 instructions), whereas checking for a 0 word is much cheaper (3 instructions). An average receptacle of 64 events is only 26 bytes long. Since the average size of a StarOS object is approximately 1800 bytes, this represents a memory overhead of less than 2%. Sensors are implemented as macro calls which expand into inline code (see section 8.2.1).

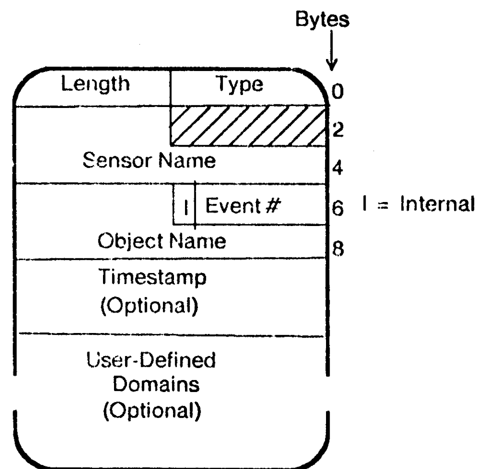

Figure D-4: Structure of an Event Record

When a sensor is invoked, it first checks the data word in the object to determine if there is a receptacle (since the macro invocation takes place within the type manager for the object containing the receptacle, the position of the receptacle, i.e., the offsets for the data word and the capability, is known). Determining that there is no receptacle requires 14.7 microseconds (equivalent to two store operations). If there is a receptacle, then it is made addressable and the enable bit for that event is tested. If the event was disabled, the sensor took 165 microseconds, equivalent to 23 store operations. This time depends strongly on how often the sensor executes, due to interactions with cached object descriptors. It is anticipated that receptacles will reside in the descriptor cache most of the time.

Otherwise, the event is enabled, and the event record is written into the pipe as described above. If there are no addresses on the stack (indicating that the pipe is full), the sensor can choose to busy wait or to discard the event record, (indivisibly) incrementing the *MissedEvents* field of the pipe or the receptacle. Note that it is unnecessary to store all three names (performer, object, initiator) in the event record, since one of these names will always be identical to the *Name* stored in the receptacle. The resident monitor is responsible for placing this name in the event record before sending it to the remote monitor.

In order to reduce the overhead of the sensor macro, and to take advantage of the multiple processors in Cm*, the formatting, storage, and transmission of the event record is broken into two stages, the first performed by the sensor macro, and the second performed by a

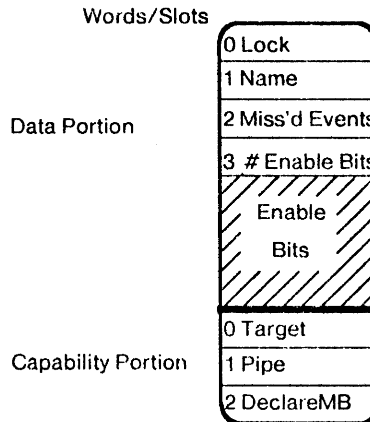

Figure D-5: The Structure of a StarOS Receptacle

process of the resident monitor asynchronously with the execution of the sensor. The sensor macro is responsible for checking the enable switch and writing the event record into the pipe's event record buffer. The resident monitor is responsible for removing event records from all the pipes, collecting them into packets, and handling the protocol for transmitting these packets across the network to the remote monitor. Since the type manager and the StarMon Accountant are probably running on different processors, the overhead of event generation is only that incurred by the sensor, which is optimized to the possible detriment of StarMon's efficiency. This arrangement is possible because StarOS allows shared memory, specifically, the event buffers in the pipes, between processes in different task forces (the sensor and StarMon may be in different taskforces). Although Medusa, the other operating system on Cm*, does not allow shared memory between processes in different task forces, it does provide a pipe mechanism which can be used to send events from the task force to the resident monitor.

The store event record operation provided by the resident monitor is implemented as the sensor macro, and the access receptacle operation provided by the type manager is implemented by passing the word and capability offsets to this macro (see section 5.2). Note that these optimizations in the event storage mechanism do not constitute typing violations; rather, they imply efficient implementations of operations on a typed object. The type managers are *not* required to be separate processes; in this case, the operations of the resident monitor on receptacles are implemented partially as a macros within other type managers, and also as code in a collection of processes. Although these changes are consistent with the type model, they have altered the semantics of the store operation: instead of the

store event record operation of the resident monitor being invoked with the event record data as parameters, the event record is stored in the pipe referenced by the receptacle at the time the event occurred, and asynchronously removed by the resident monitor at a later time. The primary advantage is that as little work as possible is performed synchronously with the execution of the operation being monitored, lowering the overhead of monitoring that operation; the disadvantage is that an arbitrary amount of time elapses between the occurrence of the event and the recording of the event at the resident monitor. If this time lapse is unacceptable, then an *event notification* mechanism must be devised. Event notification may also be necessary to inform the StarMon Accountant that a pipe is becoming too full.

Several notification actions were considered in the StarOS environment, including sending a data message to a notification mailbox, sending a capability message containing the relevant receptacle to the mailbox, invoking a StarMon process, or setting a field in a shared data structure. The final strategy adopted for notification shares characteristics with each of these alternatives. The StarMon Accountant continuously cycles through the pipes, checking to see if any new event records were added. Since this check is very fast (the StarMon Accountant simply tries to pop an address off the queue of each pipe), the delay is comparable to the delays induced by the alternatives listed above. All of these notification mechanisms fail if the event generation rate is high, since, in that case, the StarMon Accountant will be continually struggling just to keep up with the sensors, emptying the pipes before the event record buffers become full. Ultimately, determining whether to use the resident monitor invocation or the storage in receptacle/notify resident monitor mechanism is used depends on the relative efficiencies of the various operations under the given operating system. The experience with StarOS and Medusa has indicated that the time lapse between event occurrence and notification was not an important issue in most cases, and was acceptably short in the cases where it did matter.

There is one further advantage to the partitioning of the event storage operation into a synchronous and an asynchronous portion: self-monitoring is permitted. For instance, sensors may be placed in StarMon, even within the code which assembles packets and handles the protocol with the remote monitor, without triggering the generation of an infinite number of event records. Since a sensor does not invoke a StarMon process when storing an event record, it is possible for other parts of the operating system, in particular the scheduler, to treat the monitor as any other process, and to generate event records concerning the behavior of that process. These event records will be identical to other event records in format and potential for manipulation.

D.5. A Microcoded Sensor

A microcoded version of the sensor was designed to determine the maximum space and time efficiency possible on Cm*. The resulting algorithm is (intentionally) similar to the other microcoded operations in StarOS, in general trading longer execution time with smaller microcode storage requirements and shorter programmer time [Jones&Gehringer 80]. It would take an experienced programmer familiar with the current StarOS microcode an estimated two months to implement and test the version presented below [Vegdahl 82].

This section will merely sketch the design of the microcoded version in order to give the reader an impression of how the operation would work. The instruction is invoked (as are most of the StarOS operations) by writing the address of a *parameter block* to a particular address. This parameter block is assembled by a Bliss/11 macro at compile time and need not be the concern of the programmer. The components of the parameter block are shown in Figure D-6. The first two locations are set on exit from the instruction. The remainder of the parameters are identical to the ones in the other version of the sensor (all versions have identical semantics--they all write identical event records into the appropriate pipe). The *Receptacle* parameter contains the address of an object, as well as the offset of the receptacle capability within the object. The *DomainTypes* component encodes the types of all the domains, with four possible types per domain. It and the *DomainValues* may be omitted if there are no domains.

The size of this version shown in Table 8-1 is the sum of the size of the parameter block and the size of the code necessary to load the domain values into the parameter block and invoke the microcode.

The time required by this operation is harder to derive. For existing microcoded operations previously measured, it is possible to predict the execution time by simply multiplying the number of memory references executed by the operation by 10 microseconds [Jones&Gehringer 80]. Applying this approximation to the microcoded operations most similar in complexity to the sensor operation results in calculated execution times within 5% of the actual in all but one case.

A crucial assumption in this approximation is that the execution time is evenly split between external memory references (at 4.7 microseconds each) and internal processing. The primary indication that this may not be the case with the sensor operation is the complexity of the operation word in the parameter block, which requires extensive unpacking. Note that there is not a simple space-time trade-off here, because an attempt to reduce the execution time by placing the fields in separate words in the parameter block, thereby increasing the space requirements, might actually *increase* the execution time due to an increased number of memory references to access the parameter block.

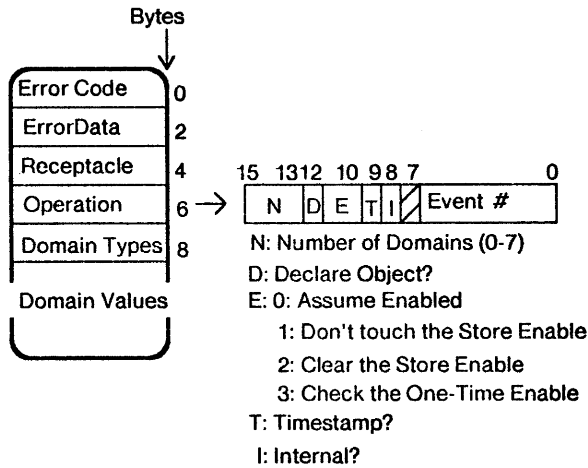

Figure D-6: The Parameter Block for the Sensor Instruction

D.6. Efficiency

To determine the relative costs of the individual features of the sensor operation, a variety of sensors were specified and measured. Each sensor was run at least 10,000 times on a dedicated processor. The code was changed slightly so that the pipe never filled up, in effect isolating the efficiency of the sensor from the rest of StarMon. All the sensor code was local to the processor, and multiplexing of the processor among several processes was disabled. In addition, constraints on the address space were relaxed to allow the sensors to execute as efficiently as possible (this is essentially another space-time tradeoff). The complete sensor descriptor file for the test sensors is shown in Figure D-7.

```
(comment Sensor definitions for determining speed of sensor calls)

(taskforce (name RSTest)
        (Simonfilename RSSimon)
)

(sensorprocess (name RSProcess)
        (requirefilename RSPSen)
        (rsslot TestSlot)
        (clockpage ClockPage)
)

(objecttype (name RSObject)
        (rsslot 0)
        (wordoffset 0)
        (requirefilename RSOSen)
)

(event (name Shortest)
        (location RSProcess)
        (minortype sensortraced)
        (assumeenabled t)
        (waittime -1)
        (spacetimeratio 0)
)

(event (name Short)
        (location RSProcess)
        (minortype sensortraced)
        (waittime -1)
        (domains (domain (name ADomain)
                         (type Integer))
        )
        (spacetimeratio 0)
        (timestamp t)
)

(event (name Medium)
        (location RSProcess)
        (minortype sensortraced)
        (waittime -1)
        (domains (domain (name ADomain)
                         (type Integer))
                 (domain (name DblDomain)
                         (type DoubleInteger))
        )
        (spacetimeratio 0)
        (timestamp t)
)

(event (name Long)
        (location RSProcess)
        (minortype sensortraced)
        (waittime -1)
        (domains (domain (name ADomain)
                         (type Integer))
                 (domain (name DblDomain)
                         (type DoubleInteger))
                 (domain (name StrDomain)
                         (type String))
        )
        (spacetimeratio 0)
        (timestamp t)
)
...                                                        rest of the file
```

Figure D-7: Sensor Description File for Measuring Sensors

Appendix E
An Extended Example

The purpose of this appendix is to illustrate the steps required to monitor an application program. Although the implementation of the monitor is far from complete, this appendix will attempt to demonstrate that the major functions have been implemented, and that it is possible to monitor an application using the relational paradigm. A second purpose is to stress the ease of installing sensors and retrieving and computing high-level monitoring information.

The steps necessary to monitor an application are as follows:

- create a sensor description file (SDF);
- send the SDF through the preprocessor (DFPre) to obtain several intermediate files;
- make minor changes to the source code of the application program to add the sensors;
- compile, link, and load the program onto Cm*;
- invoke the remote monitor; and
- present the remote monitor with a query, which is then compiled into an update network and executed.

All relevant files and terminal sessions will be reproduced here. It is important to note that these listings have not been edited or altered, except as explicitly noted. The reader can be assured that nothing is going on "behind the curtain". Unfortunately, this decision implies that various peculiarities will be present in the displays. Some of the peculiarities are the result of features irrelevant to the present discussion; others could be eliminated through straightforward changes in the code. The reader is requested to be patient and to ignore all manner of quirks in the listings.

E.1. Sensor Description File Processing

The application to be monitored, the partial differential equation program, or PDE, has already been discussed in section 3.7. The PDE is a collection of Bliss/11 [Wulf *et al.* 75b] and Task [Jones&Schwans 80] code. The Bliss/11 portion contains the algorithms to be used in the individual processes, and the Task portion specifies how data structures are to be shared among the processes and where the components are to be placed among the processors in Cm*. Task is currently specific to the StarOS operating system [Jones *et al.* 78]; there is a somewhat analogous language and compiler for the Medusa operating system [Ousterhout *et al.* 80]. For the remainder of this discussion, the use of StarOS will be assumed.

The first step in monitoring an application is to determine what sensors are desired, where these sensors are to be placed, and what attributes each sensor should have. This information is placed in a sensor description file (see section 8.2.1). Figure E-1 contains the complete listing of the sensor description file (SDF) for the PDE. This file is generated by the user.

```
(taskforce (name PDE)
        (simonfilename pdesen)
)

(sensorprocess (name PDESolver)
        (functionnumber 2)
        (requirefilename solsen)
        (rsslot SPLastCapa)
)

(event (name Iteration)
        (location PDESolver)
        (timestamp t)
        (domains (domain (name iternum)
                         (type integer))
        )
        (minortype sensortraced)
)
```

Figure E-1: PDE Sensor Description File

The SDF for the PDE contains descriptions for three objects. The *taskforce* object has two attributes: a name (PDE) and a simonfilename (pdesen). The latter is the name of the file where the remote description for this SDF will be placed (see below). There is one *sensorprocess* object for each process containing a sensor. The requirefilename attribute specifies the name of the require file to be generated by DFPre, containing Bliss macros for the sensors contained in the PDESolver process. The other two attributes are specific to StarOS and will not be discussed here. The *event* object describes a particular sensor called Iteration, located in the PDESolver process. This sensor will generate event records containing a timestamp and one user-defined integer domain called iternum. A minortype of sensortraced specifies that this sensor will be traced (as opposed to sampled) and that it will be enabled by a switch in the receptacle associated with the process containing the sensor.

The SDF contains virtually all the information required by the monitor. It is read by DFPre, which produces a remote description file and one or more require files (see Figure 8-5). DFPre is a MacLisp program running on a DECSystem-10 (all software development for Cm* is done on the DECSystem-10, with the object code sent to Cm* over the Ethernet). The dialog between DFPre and the user is illustrated in Figure E-2. As with all dialogues in this appendix, responses by the user are underlined.

After initializing itself, DFPre asks for the description file type. The "s" response instructs DFPre to read the description file format for StarOS sensor descriptions; a portion of that file is shown in Figure E-4. The name of the sensor description file is requested. The SDF is read in, with DFPre printing the names of the objects as they are encountered. The require file print command results in the generation of require files, one per sensor process (the names of the require files appear in the SDF as requirefilename attributes). The OBJ output command results in the generation of the remote description (whose name is the value of the simon-filename attribute).

The one require file produced from the PDE sensor description file is shown in Figure E-3. The primary component of this file is the definition of the ITERATIONSENSOR macro, defined to be a call of the highly parameterized StarMonSensor macro. The details of this macro are not relevant here; the point to be made is that by preprocessing the SDF, a sensor can be exactly configured to its specification in the SDF.

E.2. Description File Formats

As was mentioned both in the previous section and in section 8-5, the allowable classes and attributes for a sensor description are defined in a *description file format (DFF) file*, read by DFPre during its initialization. The DFPre file for StarOS sensor descriptions is shown in Figure E-4. The important aspects to note are

- The syntax is identical to other description files.
- For each object (e.g., taskforce, event) defined, there are two groups of attributes specified, input attributes (those present in the input file), and output attributes (those written to the remote description), along with their types.
- DFPre can be configured easily by simply modifying the corresponding DFF file.

DFPre is similar to a language-independent parser, which is given the grammar for a language (c.f., the DFF file) and a program in that language (c.f., the sensor description file), producing a parse tree of the program (c.f., the remote description).

```
r maclsp
MacLisp for MONITOR
                                                     initializing
*(preprocess)                                        run the preprocessor


Description File Preprocessor

Description File Type: M(edusa sensor) S(tarOS sensor) T(ask name)
        H(elp) Q(uit): s
Initializing description file format specification ...

Name of sensor description file: pde
Processing DSK:PDE.SDF[X335SMON] ...

PDE                                                  DFPre is reading the SDF
PDESOLVER
ITERATION

Resolving forward references ...

Command: N(ew file) O(BJ output) R(equire file print) S(ummary print)
        H(elp) Q(uit): s

Summary statistics:

Number of sensor locations: 1.
Number of object types: 0.
Number of events: 1.

Sensors:
        PDESOLVER:
                (ITERATION)

Object Types:


Command: N(ew file) O(BJ output) R(equire file print) S(ummary print)
        H(elp) Q(uit): r
Producing require files ...
Producing DSK:SOLSEN.DFS[X335SMON] ...

Command: N(ew file) O(BJ output) R(equire file print) S(ummary print)
        H(elp) Q(uit): o

Producing DSK:PDESEN.M11[X335SMON] ...

Command: N(ew file) O(BJ output) R(equire file print) S(ummary print)
        H(elp) Q(uit): q
.
```

Figure E-2: Dialogue with DFPre

```
SWITCHES NOLIST;
!This file defines the sensors for the PDESOLVER activity.

require rs.dfs[x335smOn];
external RSExists;

MACRO
MakeInternalRS(Pipe,Window,TempSlot) =
        (EXTERNAL MakeReceptacleSet;
        MakeReceptacleSet(0,RSExists,SPLASTCAPA,Pipe,1,Window,TempSlot)
        )$,
ITERATIONSENSOR(Window,N0007) =

          StarMonSensor(0,RSEXISTS,SPLASTCAPA,1,1,1,2,0,1,1,100,0,0,Window,INTTYPE,N0007)$,
NumberOfSensorEvents=1$;
SWITCHES LIST;
```

Figure E-3: Definitions File Produced From PDE SDF

E.3. Changes to the Source Code

In addition to writing an SDF, the user must also make a few minor changes to the source code. Three lines must be added to the Task portion: one specifying that a remote description (placed in the file named PdeSen by DFPre) is to be included in the task force,

```
SensorDescription: New Basic (Source=("PdeSen<C1>"));
```

and two specifying that code for creating receptacles (the MakeRS routine) should be included in the process,

```
"MakeRS[X335SMON](GO)",
```

and

```
"MakeRS[X335SMON](CP)",
```

Minor changes to the task compiler and process creator in StarOS would obviate the need for these modifications.

Three lines must also be added to the Bliss portion of the PDE. The first line is added at the beginning of the program, instructing the compiler to read the require file generated by DFPre:

```
require SolSen.DFS;
```

The second line is placed in the initialization code for the process, causing a receptacle to be created for the process:

```
MakeInternalRS(1, LocalWindow, SPWorkWindow);
```

This line could be eliminated through a minor modification in the process creator. The third line is the sensor itself, placed in the desired position, in this case, at the end of the iteration loop:

```
IterationSensor (LocalWindow, .niter[.pci]);
```

LocalWindow is a temporary storage location for use by the sensor, and .niter[.pci] is the value for the user-defined domain called iternum. At this point, the program is compiled and linked into an executable load module, ready to be loaded into Cm*.

```
; Sensor Descriptor File (SDF) format specification for StarOS

(general (filedefault ".sdf")
        (procnodefunction StarOSprocnode)
        (commandfunction StarOSDoCommand)
)

(fieldtypes (domaintype string integer doubleinteger)
        (mtype objecttraced sensortraced receptaclesampled messagesampled
                invocationsampled)
)

(input (specifies taskforce)
        (name atom)
        (simonfilename atom)                    ; default: name
        (version fixp)                           ; default: 0
        (documentation anything)
        (doc anything)
)

(output (specifies taskforce)
        (systemname string)
        (year integer)
        (month integer)
        (day integer)
        (hour integer)
        (minute integer)
        (version integer)
        (sdfilename string)
)

(input (specifies event)
        (name atom)
        (location sensorprocess)
        (object objecttype)
        (timestamp boolean)
        (domains domain)
        (declareobject boolean)
        (minortype mtype)
        (assumeenabled boolean)
        (checkonetime boolean)
        (assumeonetime boolean)
        (spacetimeratio fixp)
        (waittime fixp)
        (documentation anything)
        (doc anything)
)

(output (specifies event)
        (location sensorprocess)
        (object objecttype)
        (timestamp boolean)
        (domains domain)
        (enableindex integer)
        (minortype attributename)
        (internal boolean)
)
...
```

remainder of the file

Figure E-4: The Description File Format (DFF) File for StarOS Sensor Descriptions

The changes to the code for the PDF were examined here in such detail to emphasize how little needs to be done to monitor a program. The use had to write an SDF (16 lines long) and add 6 lines to the program, 4 of which were functionally unnecessary. A highly efficient sensor (discussed in appendix D.6) and a remote description of the SDF were created automatically by DFPre, and the remote description loaded with the rest of the code.

Before discussing how the rest of the monitor interfaces with the running program, we must examine how the monitor is started up.

E.4. Initializing the Monitor

As described in section 8.1, the monitor consists of two components: a remote monitor, executing on a Vax, and a resident monitor, executing on Cm*. The remote monitor is called *Simon*, and the resident monitor for StarOS is called *StarMon*. When StarMon is invoked, it immediately goes into a quiescent state, awaiting a packet from Simon. This behavior is consistent with the steady-state situation, where StarMon is the slave and Simon the master of the communication protocol (see section 8.4).

Figure E-5 illustrates the initialization sequence from the viewpoint of Simon running on a Vax. Recall that Simon is composed of three communicating processes: the parser, the TQuel compiler and update network, and the remote accountant (see Figure 8-2). Ideally Simon would present a unified user interface. However, to ease the task of debugging Simon, the process containing the compiler and update network also accepts commands from the user (the remote accountant was debugged indirectly through this process).

The *simonupdate* command starts up the latter two processes. The new prompt (an integer) indicates the user is now talking the FranzLisp. The initialization sequence is started with the *initacc* command. First, the remote accountant is located through interprocess communication (IPC) calls, and a packet is sent to StarMon. StarMon responds with an acknowledgement, followed by event records identifying the modules of the operating system. The command *run 1 m* instructs Simon to process incoming event records for one minute, and then wait for further commands from the user. A modification to Simon allowing it to wait for event records and user commands (via the parser process) concurrently is relatively straightforward.

Some of the event records cause Simon to display a message. For example, the message

```
[The Loader component of the catalogue 41552 is 41414.]
```

indicates that Simon received an event record stating that a component of the catalogue 41552 (the external name of the StarOS task force, implemented as a catalogue of operating system modules) with the string name "Loader" is the object with the external name 41414.

```
[shell] simonupdate
...                                                    messages from FranzLisp
1. initacc 0
[Trying to locate Simon Accountant ...]
[Located Simon Accountant.]
[Waiting for Ethernet connection with the resident monitor...]
[Connection established.]
[Processing the operating system taskforce(s) ...]
[The Catalogue-Module component of the catalogue 41552 is 41389.]
[The FTP-Module component of the catalogue 41552 is 41539.]
[The Garbage-Collector component of the catalogue 41552 is 41367.]
[The Loader component of the catalogue 41552 is 41414.]
[The Nucleus component of the catalogue 41552 is 41248.]
[The Object-Manager component of the catalogue 41552 is 41230.]
[The Process-Module component of the catalogue 41552 is 41240.]
[The PUP-Module component of the catalogue 41552 is 41499.]
[The Reconfiguration component of the catalogue 41552 is 41259.]
[The Sensor-Description component of the catalogue 41552 is 41544.]
[The Six12-Debugger component of the catalogue 41552 is 41491.]
[The SENSOR description for the taskforce 41552 is 41546.]
[Defining the TASKFORCE STAROS.]
...                                         other objects and domains are defined
[Defining the DOMAIN INDEX.]
2. run 1 m
[CheckPoint received: Time: 275951.]
[Initializing event number 1: DESCRINTEGER.]
[Initializing event number 2: DESCRSTRING.]
[Initializing event number 3: DESCREND.]
[Initializing event number 4: COMPONENTSTRING.]
[Initializing event number 5: COMPONENTINTEGER.]
[Finished the STAROS taskforce SENSOR definitions.]
[The Rick component of the catalogue 41555 is 41581.]
3. showstatus
Status of the Remote-Resident Monitor Interface

There were 141 event records processed.

The defined task forces are:
Taskforce tf-00014:
STAROS Sensor Description File:
(Definition in DSK:STAROS.SDF[X335SMON] (version 1),
    processed on July 18, 1982, at 15:12):
        DESCRINTEGER ((ATTRIBUTE: INTEGER) (OBJECT: INTEGER) (VALUE: INTEGER) )
        DESCRSTRING ((ATTRIBUTE: INTEGER) (OBJECT: INTEGER) (VALUE: STRING) )
        DESCREND (Timestamped) ()
        COMPONENTSTRING (Timestamped) ((COMPONENT: DOUBLEINTEGER) (NAME: STRING) )
        COMPONENTINTEGER (Timestamped) ((COMPONENT: DOUBLEINTEGER) (INDEX:
  INTEGER) )
Components:
        Catalogue-Module 41389
        ...                                        rest of the components
        Six12-Debugger 41491
```

Figure E-5: Initialization Dialogue from Simon's Perspective

After the components of the operating system catalogue have been sent, StarMon sends the remote description for the operating system itself. Finally, the components of the catalogue of users is sent, indicating one user. The *showstatus* command illustrates that one task force has been defined (the StarOS task force), containing five sensors and eleven components. This description was in fact bootstrapped over the Ethernet: the first three sensors listed were the ones which sent the description in the first place. Note that the initialization sequence required 141 event records to be transferred to Simon from StarMon.

```
4. (GetComponents Rick)
1
5. run 10 s
[The Library component of the catalogue 41581 is 41552.]
[The M component of the catalogue 41581 is 41647.]
[The P component of the catalogue 41581 is 41814.]
[The User component of the catalogue 41581 is 41555.]
6. (GetComponents P)
1
7. run 30 s
[The PDEModule component of the catalogue 41814 is 41787.]
[The SensorDescription component of the catalogue 41814 is 41786.]
[The SENSOR description of the taskforce 41814 is 41786.]
[Defining the TASKFORCE PDE.]
...                                     other objects and domains defined
[Initializing event number 6: ITERATION.]
[Finished the PDE taskforce SENSOR definitions.]
8. showstatus
Status of the Remote-Resident Monitor Interface

There were 196 event records processed.

The defined task forces are:
Taskforce tf-00014:
STAROS Sensor Description file:
    Catalogue-Module 41389
    ...
    Six12-Debugger 41491

Taskforce tf-00058:
PDE Sensor Description file:
(Definition in DSK:PDE.SDF[X335SMON] (version 0),
    processed on July 24, 1982, at 16:32):
        ITERATION (Timestamped) ((ITERNUM : INTEGER) )
Components:
        PDEMODULE: 41787
        SENSORDESCRIPTION: 41786
9.
```

Figure E-6: Retrieving the Information Concerning the PDE

At this point, the initialization sequence has completed, and Simon knows about the sensors located in the operating system, including those sensors within StarMon. The PDE is loaded into StarOS, and the dialogue shown in Figure E-6 is followed. The components sensor is sampled by giving a *GetComponents* command; the information returns in subsequent event records, as shown. The argument of this command is the external name for a catalogue, in this case, the user catalogue associated with Rick (object 41581). This catalogue contains, among other things, all modules loaded by this user. Since the PDE had been loaded under

the name "P", its components are requested. StarMon notices that one of the components has the name "SensorDescription"; it assumes the component contains a remote description, and it automatically sends the information in this component to Simon. The *showstatus* command now lists two task forces, the StarOS task force and the PDE task force. Since the PDE taskforce is less complex, only about 50 event records were required to send its description.

E.5. Query Processing

Simon now knows about two taskforces containing six sensors. Simon fully supports the conceptualization of a temporal database, so these sensors correspond to primitive relations which can be referenced in TQuel queries. Since the ITERATION sensor is sensortraced, there is an associated primitive period relation (recall that traced events are automatically converted to period relations consisting of two domains, the process containing the sensor, and one user-defined domain):

```
ITERATION (PROCESS ITERNUM)
```

This relation is used in the example queries in section 3.7, written as a macro in Figure E-7. This macro has two parameters, specifying the external names for the processes participating in the query. The backslash at the end of each line indicates that the macro extends to the next line (macros are usually terminated by the end of the line).

```
{define; examplequery $1 $2;                        \
range of A is Iteration                             \
range of B is Iteration                             \
retrieve AOverB (Diff = B.IterNum - A.IterNum)      \
where A.Process = $1 and B.Process = $2             \
        and A.IterNum > B.IterNum + 1               \
                                                    \
range of AB is AOverB                               \
retrieve Over (Percent = AvgC(AB) * 100)            \
                                                    \
retrieve Catch                                      \
where A.Process = $1 and B.Process = $2             \
        and A.IterNum = B.IterNum                   \
when A.Start ; B.start                              \
at B.start                                          \
                                                    \
display Over, Catch                                 \
}
```

Figure E-7: PDE Query Contained in the file demoquery

This query is parsed by the process invoked by the *simonmonitor* command (see Figure E-8). This program was derived from the terminal interface and parser for Ingres. It therefore shares many nice features with the Ingres system, including a command to invoke the editor, a complete macro facility, command files, etc. Commands are preceded with a backslash; the "\t" command prints the day, date, and time; the "\s" command escapes to the Unix shell, the command interpreter [Bourne 78] where other programs can be run; and the "\i" com-

mand reads a file (in this case, the file containing the examplequery macro) into the workspace. Text not preceded by a backslash (e.g., examplequery 41521 41653) is inserted directly into the workspace. 41521 and 41653 are external names of PDE processes; these names were obtained by ad hoc means[28], although in the future there will be a sampled sensor in StarMon for finding the processes associated with a specified task force. The "\g" command causes the contents of the workspace to be evaluated and a parsetree written to the file named parsetree, and the "\q" command causes an exit from the program. The parsetree file (containing 74 nodes) is then displayed on the terminal (the cat program in Unix displays one or more files on the terminal), and the shell is exited using the ↑D key, bringing us back to the process containing the tquel compiler and the update network.

```
9. shell                                        invoke the shell (the user interface for Unix)
[shell] simonmonitor
SIMON version 0.2 (7/26/82) login
Wed Jul 28 19:58:43 1982
go
*\t
Wed Jul 28 19:59:10 1982
*\i demoquery
continue
*examplequery 41521 41653
*\g
Executing . . .

continue
*\r
go
*\q
SIMON version 0.2 (7/26/82) logout
Wed Jul 28 20:01:21 1982
goodbye rts -- come again
[shell] cat parsetree
(2 12 13 ITERATION 0 0)
(3 0 35 A 2 0)
...                                             the rest of the parsetree
(75 0 34 nil 74 0)
[shell] ↑D
10.
```

Figure E-8: Parsing the Query

The parsetree file contains the parse tree in a linearized form. The nodes in the tree are listed one per line. The first integer is the node's index; the second integer specifies the type of node; the next two values specify auxiliary information in the node, and the last two integers specify the indices of the left and right sons of the node. This file is generated by a bottom-up scan of the parsetree, and is similar in use to the LG (linear graph) files developed in the PQCC project [Leverett *et al.* 80].

---

[28] The external names of processes were obtained when the PDE task force was invoked on Cm*. Each process created its receptacle and sent it to StarMon, which then extracts the external name of the process from the receptacle and displays it on the terminal.

The TQuel compiler is invoked with a *processquery* command in FranzLisp, resulting in an update network for the query (see Figure E-9). This command reads the node descriptions from the file named parsetree, constructs an internal parsetree, and generates an update network for the query. This update network is specified by a sequence of primitive constructor functions (createaccess, createop, and link--see section 6.5). Since automatic enabling of sensors has not yet been implemented, the correct sensors must be enabled by the user through commands on the Vax. The *enableevent* command takes two arguments, a sensor name and an external object name. There is an analogous *disableevent* command[29].

To illustrate tuples flowing into the network, a few primitive constructor functions are executed by the user. The PDE is then started, causing event records to be sent across the Ethernet and converted into tuples which flow through the update network until they encounter a display operator node, causing a message to be printed on the terminal. Finally, the sensors must be disabled by the user through commands on the Vax.

The actual event records flowing over the Ethernet from Cm* were not interesting, so a set of 25 event records were constructed to produce interesting results. The fakeaccess functions (see Figure E-10) were loaded, and the testupdate command executed to read the test event records from a file and send them through the update network. The input tuples resulted in 7 output tuples. Recall that Catch is an event relation with no explicit domains, and that Over is a period relation with one temporal domain called Percent. The value of this domain at, say, time 300 can be derived from the tuple valid during the time [285-316]:

$$\text{Percent} = (10000 + 100 * (300-285)) / (277 + 1 * (300-285))$$
$$= 39.4\ \%$$

The last few lines provide statistics used in analyzing the performance of the update network. The results of this analysis are discussed in the final appendix.

[29] The external object names currently must be given as signed integers. Hence, the external object name 41521 is entered as -24015.

```
10. processquery
Processing statement 1...                                  range of A is...
Processing statement 2...                                  range of B is...
Processing statement 3...                                  retrieve AOverB...
[(createaccess ITERATION access-00377)]
[(createop eventtoperiod etop-00378 (1 4))]
[(link access-00377 etop-00378 left)]
...                                                        Other constructor functions
[(link applyop-00390 projection-00392 left)]
Processing statement 4...                                  range of AB is...
Processing statement 5...                                  retrieve Over...
[(createop aggrop opAVGC-00393 (opAVGC -1 -1 -1 3 0 1))]
...
[(link applyop-00398 projection-00400 left)]
Processing statement 6...                                  retrieve Catch...
[(createaccess ITERATION access-00404)]
...
Processing statement 7...                                  display Over...
...
nil
11. enableevent ITERATION -24015
[Enabling event ITERATION (taskforce: PDE class: 2) for object 41521.]
12. enableevent ITERATION -23883
[Enabling event ITERATION (taskforce: PDE class: 2) for object 41653.]
13. (createaccess ITERATION a-1)
{(&record . nodeinstance) ... }
14. (createop display d-1 (ITERATION))
{(&record . nodeinstance) ... }
15. (link a-1 d-1)
nil
16. run 30 s
Display (d-1): ITERATION: (ITERNUM)
<tuple [2908341- ]: event
((2 : 41521) (3 : 0) (4 : 1))>
Display (d-1): ITERATION: (ITERNUM)
<tuple [2908353- ]: event
((2: 41653) (3 : 0) (4 : 1))>
...
Display (d-1): ITERATION: (ITERNUM)
<tuple [2911736- ]: event
((2 : 41653) (3 : 0) (4 : 40))>
17. disable ITERATION -24015
[Disabling event ITERATION (taskforce: PDE class: 2) for object 41521.]
t
18. disable ITERATION -23883
[Disabling event ITERATION (taskforce: PDE class: 2) for object 41653.]
t
```

Figure E-9: Compiling and running the update Network

```
19. load fakeaccess
t
20. testupdate 41521 41653
[Enqueuing fake data from thesisdemo/testdata.]
Start: (6146 0) Display (d-1): ITERATION: (ITERNUM)
<tuple [0- ]:event
((2 : 41521) (3 : 0) (4 : 1))>
...
Display (d-1): ITERATION: (ITERNUM)
<tuple [838- ]: event
((2 : 41653) (3 : 0) (4 : 26))>
Display (display-00433): CATCH: (START)
<tuple [0- ]: event
nil>
Display (display-00433): CATCH: (START)
<tuple [91- ]: event
nil>
...
Display (display-00433): OVER: (START STOP PERCENT)
<tuple [0-8]: event
((3 : (0 0 1 0)))>
Display (display-00433): OVER: (START STOP PERCENT)
<tuple [8-15]: event
((3 : (100 0 1 0)))>
...
Display (display-00433): OVER: (START STOP PERCENT)
<tuple [285-316]: event
((3 : (10000 100 277 1)))>
Display (display-00433): OVER: (START STOP PERCENT)
<tuple [316-321]: event
((3 : (131000 100 308 1))>
...
Display (display-00433): OVER: (START STOP PERCENT)
<tuple [594-601]: event
((3 : (34500 100 586 1)))>
Stop: (7875 947), compute: 782
< *runq* [runqentry]: 1309 enqueues, not traced,
Length: (Initial: 10) (Delta: 10) (present: 282) (Filled: 0)>
21. exit
[shell]
```

Figure E-10: Running the Update Network Using Generated Event Records

Appendix F
Update Network Performance

This appendix provides the details of the measurements of the update network. As discussed in section 8.6.3, three sets of measurements were taken: one with the update network generated by the existing compiler, one with this network hand optimized using the strategies discussed in section 7.2, and one with Lisp functions generated by hand from the optimized network, using only strategies that could be readily implemented. All three versions correspond to the queries given in section 3.7. It should be emphasized that the measurements examined here only apply to this one set of queries, and may not be representative of queries in general. On the other hand, these queries are somewhat complex, involving <where>, <when>, <start>, and <at clause>s, as well as several <tuple variable>s and expressions. Most queries will probably be less complex.

Several measurements were taken for each version. One was the average number of node fires resulting from each input tuple, where a *fire* is defined to be the invocation of an access or operator node. Note that this value is normally greater than two, since each input tuple causes at least one access node and one initial operator node to fire. This value is related to the average depth attained by an input tuple in the network, but is not equivalent to this value, since an input tuple van trigger the generation of several new tuples, especially in the cartesian product node.

The second measurement taken is the average execution time per node fire. This time has two components: the invocation time for a node, which is independent of the node type, and the execution time of a node once it has been invoked, which is dependent of the type of the node. Given this measurement, it is easy to calculate the number of input tuples that can be absorbed each second, indicating the execution speed of the network.

Cartesian products are expensive, so the number of output tuples they produce, per tuple flowing into the node, was measured. Since several of the optimizations reflect this cost, we would expect the optimized version to show significant improvement in this regard.

Finally, the number of output tuples per input tuple was determined. This value is invariant across the three versions (given that they implement the same queries) and is more highly dependent on the particular queries (and the input data) than the other measurements. All

trials with actual input tuples resulted in few output tuples, since one process tended to get and stay ahead of the other process. Hence, an artificial input stream of 50 tuples was chosen to produce interesting results. For this stream, there were 32 output tuples produced. The other measurements using the test input tuples are quite pessimistic for this update network, since each input tuple resulting in an output tuple causes more node fires, more intermediate tuples to be produced, and generally more computation than an input tuple eliminated during the processing.

The execution times were all measured using the *ptime* system function, which returns two values, the cumulative runtime of the process and the cumulative runtime used by the garbage collector. Both times are in *jiffies*, that is, (1/60) seconds, or 16.67 milliseconds. The experiments were replicated many times to reduce this sampling artifact. Runtime measurements indicate performance on a dedicated Vax 11/780, and are therefore relatively invariant to current system load. Measurements involving time will be presented in the format of *a (b)*, where *a* includes the overhead of garbage collection, and *b* assumes that garbage collection is instantaneous. Both figures are given to emphasize one artifact resulting from the use of Lisp as the implementation language.

The execution times concern only the update network, and thus only a portion of the actual performance of the monitor. In particular, the remote accountant (see section 8.5), also executing on the Vax, will reduce the effective tuple rate. Informal arguments in section 8.8 indicate that the components of the monitor have comparable efficiencies.

F.1. The Unoptimized Version

The code for constructing the update network as generated by the compiler is shown in Figure F-1, and the update network itself is shown in Figures F-2 and F-3. It contains four access nodes and 24 operator nodes, of seven types. The following measurements were taken:

- 19.9 node fires per input tuple;
- 17.5 (7.6) milliseconds per node fire;
- 60% of this time spent in garbage collection;
- 2.8 output tuples generated per input tuple of a cartesian product;
- 2.9 (6.6) input tuples per second processed; and
- 77% of these tuples later eliminated.

```
Processing statement 1...
Processing statement 2...
Processing statement 3...
[(createaccess ITERATION access-00499)]
[(createop eventtoperiod etop-00500 (1 4))]
[(link access-00499 etop-00500 left)]
[(createaccess ITERATION access-00497)]
[(createop eventtoperiod etop-00498 (1 4))]
[(link access-00497 etop-00498 left)]
[(createop cartesian cartesian-00501 (0 4))]
[(link etop-00500 cartesian-00501 left)]
[(link etop-00498 cartesian-00501 right)]
[(createop selection selection-00502 ((lambda ($3 $8) (greaterp $3 (+ $8 1))) (3
  8)))]
[(link cartesian-00501 selection-00502 left)]
[(createop selection selection-00504 ((lambda ($6) (equal $6 41653)) (6)))]
[(link selection-00502 selection-00504 left)]
[(createop selection selection-00506 ((lambda ($1) (equal $1 41521)) (1)))]
[(link selection-00504 selection-00506 left)]
[(createop applyop applyop-00508 (10 (lambda ($0 $5) (stopfunc (NEXTfunc $0 $5)))
  (0 5)))]
[(link selection-00506 applyop-00508 left)]
[(createop applyop applyop-00510 (11 (lambda ($4 $9) (startfunc (NEXTfunc $4 $9)))
  (4 9)))]
[(link applyop-00508 applyop-00510 left)]
[(createop applyop applyop-00512 (12 (lambda ($3 $8) (- $8 $3)) (3 8)))]
[(link applyop-00510 applyop-00512 left)]
[(createop projection projection-00514 ((10 11 12)))]
[(link applyop-00512 projection-00514 left)]
Processing statement 4...
Processing statement 5...
[(createop aggrop opAVGC-00518 (opAVGC -1 -1 -1 3 0 1))]
[(link projection-00514 opAVGC-00518 left)]
[(createop applyop applyop-00519 (4 (lambda ($3) (productti $3 100)) (3)))]
[(link opAVGC-00518 applyop-00519 left)]
[(createop projection projection-00521 ((0 1 4)))]
[(link applyop-00519 projection-00521 left)]
Processing statement 6...
[(createaccess ITERATION access-00525)]
[(createop eventtoperiod etop-00526 (1 4))]
[(link access-00525 etop-00526 left)]
[(createaccess ITERATION access-00523)]
[(createop eventtoperiod etop-00524 (1 4))]
[(link access-00523 etop-00524 left)]
[(createop cartesian cartesian-00527 (0 4))]
[(link etop-00526 cartesian-00527 left)]
[(link etop-00524 cartesian-00527 right)]
[(createop selection selection-00528 ((lambda ($3 $8) (equal $8 $3)) (3 8)))]
[(link cartesian-00527 selection-00528 left)]
[(createop selection selection-00530 ((lambda ($1) (equal $1 41653)) (1)))]
[(link selection-00528 selection-00530 left)]
[(createop selection selection-00532 ((lambda ($6) (equal $6 41521)) (6)))]
[(link selection-00530 selection-00532 left)]
[(createop selection selection-00534 ((lambda ($0 $5) (followpred $5 $0)) (0 5)))]
[(link selection-00532 selection-00534 left)]
[(createop projection projection-00536 ((0)))]
[(link selection-00534 projection-00536 left)]
[(createop switchdisposition switchdisposition-00537)]
[(link projection-00536 switchdisposition-00537 left)]
Processing statement 7...
[(createop display display-00538 (CATCH (START)))]
[(link switchdisposition-00537 display-00538)]
[(createop display display-00539 (OVER (START STOP PERCENT)))]
[(link projection-00521 display-00539)]
```

Figure F-1: Code for the Unoptimized Update Network as Generated by the TQuel Compiler

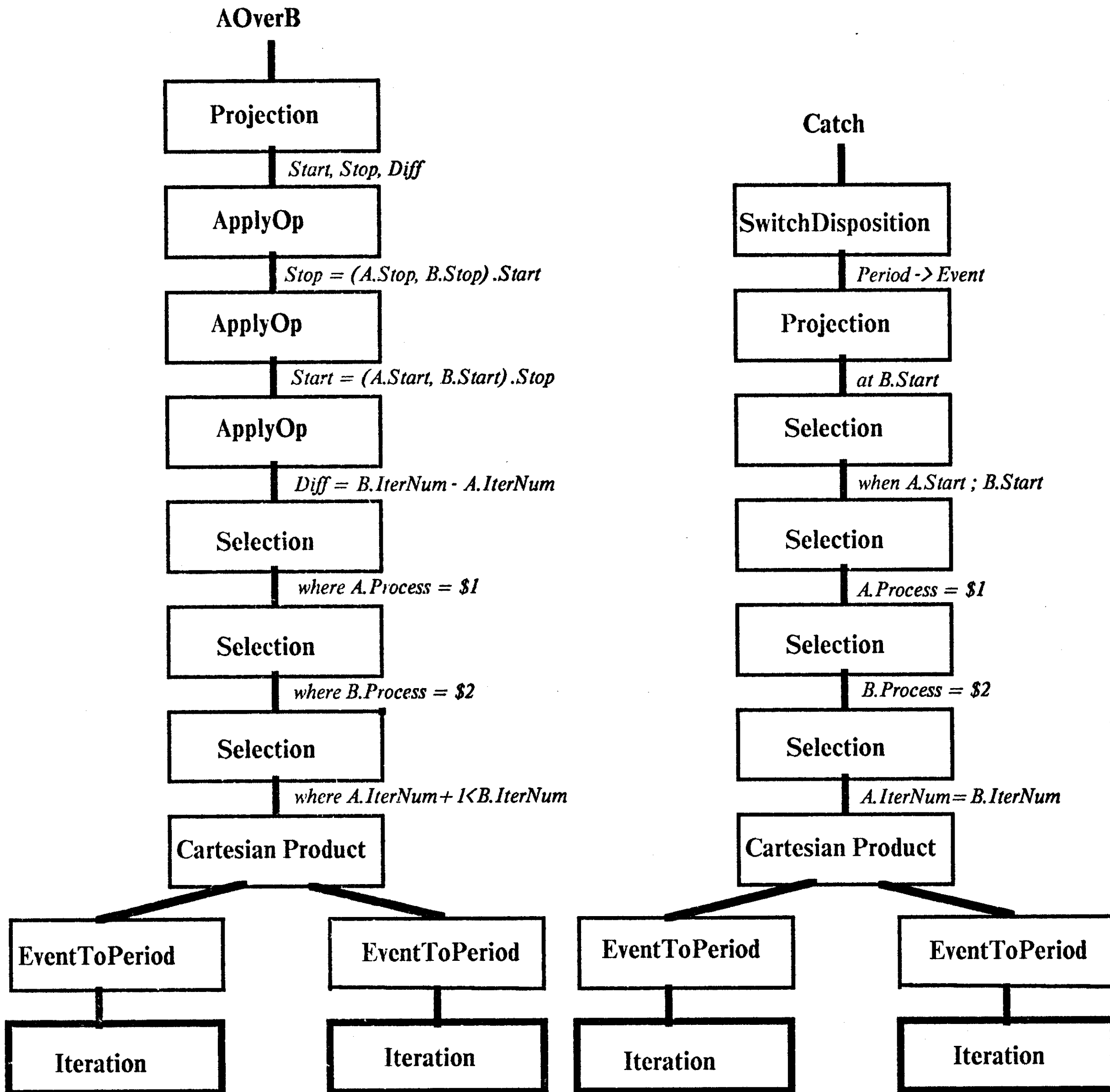

Figure F-2: The Unoptimized Update Network Generated by the TQuel Compiler, Part 1

The first measurement indicates that most of the tuples move through several nodes before they are eliminated; although the selection nodes are right after the cartesian product nodes,

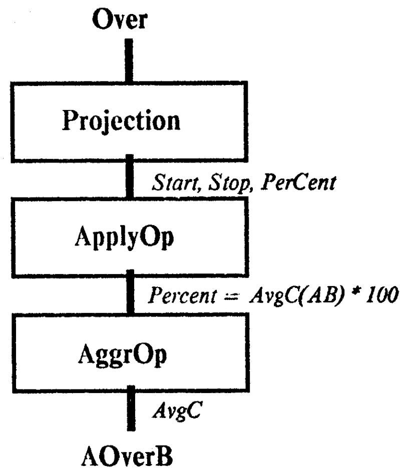

Figure F-3: The Unoptimized Update Network Generated by the TQuel Compiler, Part 2

they still are at least four nodes away from the input nodes. The number of nodes generated by the cartesian product nodes is rather high; this value, coupled with the fact that less than a fourth make it through the rest of the network, implies that the cartesian products are contributing to the low efficiency of the network. Finally, the execution time per node fire is excessive, although the performance data does not discern between the node invocation component and the node execution component of this time.

F.2. The Optimized Version

The following optimizations were performed by hand to produce the optimized version shown in Figure F-4:

1. The selection nodes were moved to just after the access nodes, thereby reducing the number of tuples flowing into the cartesian product nodes.

2. The Diff domain of the intermediate relation AOverB was eliminated through flow analysis of the domains.

3. A more efficient eventtoperiod algorithm was used, because only one process value was possible for the process domain.

4. A more efficient cartesian product could be used, because the inputs are now ordered by their start and stop times.

These optimizations are discussed in more detail in section 7.2.

The resulting update network contains two access nodes and 18 operator nodes, and has the following characteristics:

- 9.2 node fires per input tuple;
- 7.2 (4.7) milliseconds per node fire;
- 35% of this time was spent in garbage collection;
- 15 (23) input tuples per second;
- 0.9 output tuples generated per input tuple of the cartesian product;
- 28% of these tuples later eliminated.

All of these values were dramatically improved over the unoptimized version. The transfer of the selection nodes resulted in a 50% reduction in the number of node fires per input tuple. The cartesian product effectively eliminated 10% of its input tuples, rather than outputting more tuples than were input (the usual case), due to the requirement that the periods represented by the tuples overlap in order for an output tuple to be generated. The survival rate of those tuples that were output was three times that of the unoptimized version. The 40% reduction in execution time of a node fire may be attributed to two factors: the more efficient eventtoperiod and cartesianproduct algorithms and the reduced storage requirements of the cartesianproduct nodes as a result of fewer input tuples.

F.3. The Compiled Version

The optimizations listed in the previous section were effective at reducing the deleterious effects of the cartesian products. However, the execution time of a node fire is still a significant problem. Even if optimizations could lower the number of node fires per input tuple to 1, the processing rate would still be an unsatisfactory 140 input tuples per second.

As mentioned above, the node fire time has two components, the invocation time for a node and the execution time of a node once it has been invoked. Measurements were taken of an operator node that did nothing but return its input tuple, with an execution time (the second component) of a few microseconds. The average time for a node fire was 2.4 milliseconds.

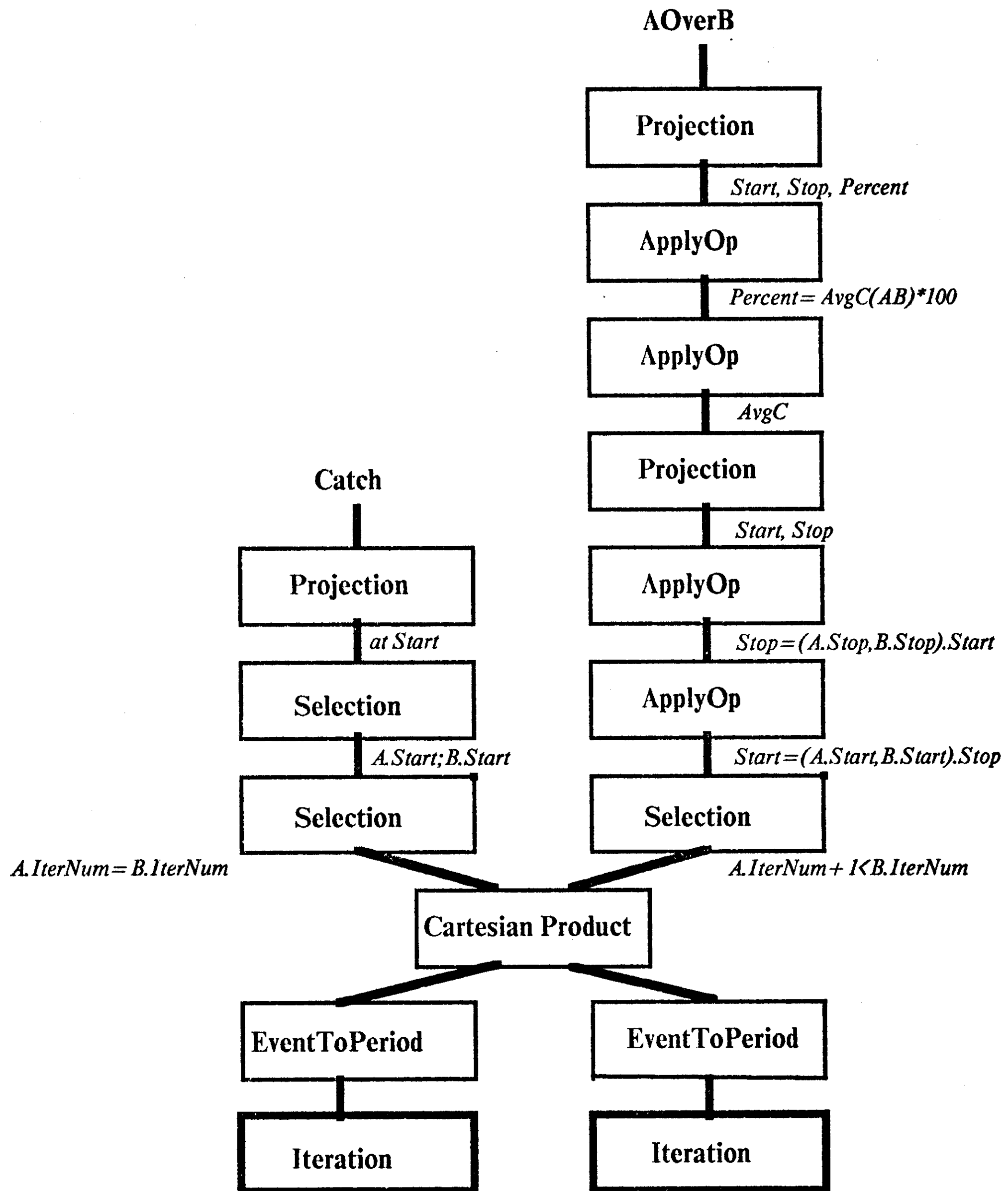

Figure F-4: The Hand-Optimized Update Network

Hence, the invocation time accounts for approximately a third, and the execution time, two thirds, of the time fire time for a node.

The following factors all contributed to the low efficiency:

- Data structures were implemented as hunks (arrays of Lisp pointers) which are expensive to allocate and to garbage collect.

- The queuing in breadth-first scheduling involved a lot of copying, requiring much allocation and deallocation.

- Operator nodes were general, and thus had to check data structures extensively to determine the specific operations to perform.

Compilation strategies were developed to eliminate these inefficiencies. The update network compiler translates a sequence of create and link operations (the update network) into a collection of Lisp functions, which are then compiled by the Lisp compiler into Vax assembly language. Conceptually, an entire update network is converted into a specialized operator node (see Figure F-5). In fact, implementing queries in this manner allows portions of the update network to be interpreted and other portions to be compiled.

An update network compiler was designed but not implemented. The techniques developed for such a compiler were tested by hand-compiling the optimized version of the previous section, and then measuring the performance of the compiled version. The following optimizations were made during the hand compilation:

1. No hunks were used; data structures were implemented as list structures or as collections of local variables.

2. Each node was implemented as a Lisp function, allowing a node to be fired by simply invoking the function.

3. Depth-first scheduling was used, eliminating a ready queue of nodes.

4. The code for each node was optimized to perform only the necessary calculations.

5. The functions that were only called once were expanded inline.

6. Nodes were invoked using the Lisp apply function, so that the arguments of the invoked function consist of the domains of the tuple, rather than the tuple itself. This technique, used in the Ops4 implementation [Forgy 79], eliminates the overhead of unpacking the tuple to access its domains.

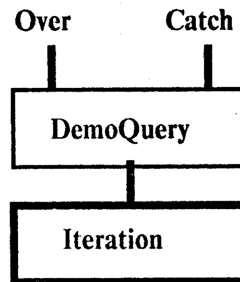

Figure F-5: The Hand-Compiled Update Network

7. The temporary storage for a node was optimized and was kept in global variables.

The performance of the hand-compiled query was impressive:

- 1 node fire per input tuple;
- 1.6 (1.5) milliseconds per node fire;
- < 10% of this time was spent in garbage collection;
- 600 (660) input tuples per second;
- 0.9 output tuples generated per input tuple of the cartesian product; and
- 28% of these tuples were later eliminated.

The execution time for the node fire is one-fourth that of the optimized version (while performing the computation of the entire network!), mirroring the high overhead of breadth-first scheduling in the latter version. The elimination of hunks and the careful use of temporary variables greatly reduced the overhead for garbage collection.

The low garbage collection overhead also indicates that the compiled version is not utilizing Lisp fully. Indeed, the primary reason for compiling the update network is to avoid the convenient features of the language which are expensive in execution time. The target language could have been C instead of Lisp. In fact, it could have been assembly language

without too much additional work, since many high level language features (e.g., structure definitions and type checking) would not be utilized anyway.

F.4. Summary

This appendix has detailed the application of a host of techniques for increasing the efficiency of update networks. These techniques, applied in concert, result in a dramatic speedup factor of over 200 for a particular set of queries, without altering the semantics of the network. A common attribute of these techniques is that they exploit knowledge concerning the update network. For example,

- Cartesian products are expensive, so move selections to below them, if possible.
- More efficient operator nodes may be used given particular tuple orders.
- If domains are not used, they need not be computed.
- Depth-first scheduling does not require a ready queue, so it can be eliminated.
- If an operator node is referenced only once in an update network, it can be compiled inline.

These specific techniques are instances of one of the steps in the iterative process outline in section 8.7:

> 4. Apply all the available information to a particular instance of the problem in order to efficiently perform the desired action, trading generality for efficiency.