A Relational Approach to Monitoring Complex Systems

December, 1985

Richard Snodgrass

Department of Computer Science
University of North Carolina
Chapel Hill, North Carolina 27514

Abstract

Monitoring is an essential part of many program development tools, and plays a central role in debugging, optimization, status reporting, and reconfiguration. Traditional monitoring techniques are inadequate when monitoring complex systems such as multiprocessors or distributed systems. A new approach is described in which historical databases form the conceptual basis for the information processed by the monitor. This approach permits advances in specifying the low level data collection, specifying the analysis of the collected data, performing the analysis, and displaying the results. A prototype implementation demonstrates the feasibility of the approach.

Table of Contents

List of Figures

List of Examples

1. Introduction

Monitoring is the extraction of dynamic information concerning a computational process, as that process executes. This definition encompasses aspects of measurement, observation, and testing.[1] Monitoring is a fundamental component of many computing activities:

- One use of monitoring is to facilitate the debugging of complex programs. Debugging proceeds in five stages [Model 1978]: (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 and, to some extent, the second and third stages in this process. Monitoring is a 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.
- 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 program once it is implemented. Tuning requires feedback on the program's efficiency, which is determined from measurements on the program while it is running.
- 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 nature of hardware and software failures.
- And finally, monitoring information may also be used internally by the application program. 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) [Ogle, et al. 1985, Rashid & Robertson 1982, Wulf et al. 1975]. Monitoring information is also valuable for programs which must be reliable; the fact that a processor (executing processes belonging to a program) has failed, for example, is important to the program if it must be able to recover from such failures.

Monitoring is thus an essential function. In one study of program development tools [Houghton 1982], a quarter of these tools were highly dependent upon monitoring, including those under the categories of tracing, tuning, timing, and resource allocation. Much has been written about monitoring on uniprocessor systems (c.f., the bibliographies [Agajaman 1975, Perlis, et al. 1981]) and the general techniques of tracing and sampling are well established.

---

[1] There are at least two other definitions of *monitor* that should be mentioned: a synonym for operating system and an arbiter of access to a data structure in order to ensure specified invariants, usually relating to synchronization [Hoare 1974]. Both definitions emphasize the *control*, rather than the *observational* aspects of monitoring. Monitoring is closely associated with, but strictly separate from, activities which change the course of the computational activity. The term monitor as used in this paper is the (usually software) agent performing the monitoring.

The term complex system used in the title is intentionally vague. We use the term here to include large uniprocessors, tightly coupled multiprocessor systems, and loosely coupled local and long haul networks. Two distinctions relevant to monitoring are that complex systems often exhibit a lack of central control and that a quantitative difference between simple and complex systems in the number of system components (processors, processes, memory, addressing domains, etc.) leads to a qualitative difference in the sophistication required of the monitor. These two aspects conspire to make monitoring a complex system a difficult (and thus interesting) task.

In this paper, we argue that an *historical database*, an extension of a conventional relational database, is an appropriate formalization of the information processed by the monitor of a complex system. This approach induces changes in the ordering of the steps performed during monitoring, as well as changes in the steps themselves. In Section 2 we examine the sequential process of traditional monitoring. The third section reviews efforts in the area of database management that address the central problem of monitoring, that of information processing. Sections 4 through 8 propose the new approach, exposing the many opportunities such an approach presents. In Section 9 we return to the traditional approach, comparing it with our approach. Section 10 briefly examines a prototype implementation, and the last two sections offer conclusions and directions for future work.

2. Traditional Monitoring

The purpose of this section is to provide an overview of monitoring as presently practiced. A few definitions are useful. The *subject system* is the software system being monitored, usually the operating system of the user's program. A *sensor* is a section of code within the subject system which transfers to the monitor information concerning an event or state within the system. If the sensor is *traced,* then a data packet is transferred to the monitor each time a particular event occurs. If the sensor is *sampled,* then a data packet is transferred each time the monitor requests the sensor to do so. This *data packet* may be as simple as a bit that is complemented when the event occurs, or as complex as a long record containing the contents of system queues. The removal of irrelevant data packets is termed *filtering.*

Implicit in most discussions on monitoring is a eight step sequential process:

Step 1: Sensor Configuration
This step involves deciding what information the sensor will record and where the sensor will be located.

Step 2: Sensor Installation
The sensors must be coded and placed in the correct location in the subject system. Provision must be made for temporary and permanent storage of the collected data.

Step 3: Enabling Sensors
Some sensors are permanently enabled, storing monitoring data whenever executed, while others may be individually or collectively enabled, usually by directives from the user.

Step 4: Data Generation
The subject program is executed, and the collected data stored on disk or magnetic tape. Generally the user has little control of the monitoring at this point.

Step 5: Analysis Specification
In most systems the user is given a menu of supported analyses; sometimes a simple command language is available.

Step 6: Display Specification
Either only one display format is available, or the user is given a menu of formats, ranging from a list of data packets printed in a readable form to canned reports to simple graphics (graphs or histograms).

Step 7: Data Analysis
The data analysis invariably occurs in batch mode long after the data has been collected.

Step 8: Information Display
Usually this step occurs immediately after data analysis, although a few packages allow the analyzed data to be displayed at a later time.

While most monitoring systems follow the sequence of phases just listed, in the precise order given (e.g., [Malone 1983, Tetzlaff 1979]), there is a variety of alternative orderings within each phase. Many systems do not differentiate between sensor configuration and sensor installation. In some systems, sensors are always enabled, so that the enabling sensors step occurs in the second step when the sensors are installed (e.g., [Bowie & Linders 1978, IBM 1984]). Some systems support only one display format, effectively combining the analysis and display specification steps (e.g., [Graham et al. 1982, McDaniel 1982, Tolopka 1981]); other systems allow the display to be specified after the data has been analyzed(e.g., [Cooperman et al. 1972, DEC 1983]).

When considering the monitoring of a complex system, the first strategy to be examined is to extend each step in obvious ways. Such an approach is problematic at every step, due to the logical and physical distribution of the monitor and the subject program(s). These difficulties are examined

in detail in Section 9, where the traditional approach is compared with our proposed approach. The next section will review related work in processing information, the basic function of a monitor, and will examine how results from this work may be applied to monitoring. Section 4 will then present a new approach to monitoring based on this analysis.

3. The Relational Model

In an abstract sense, the process of monitoring is concerned with retrieving information and presenting this information in a derived form to the user. Hence, the monitor is fundamentally an information processing agent, with the information describing time-varying relationships between entities involved in the computation.

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 1970]. The relational model provides both a structuring of the information and manipulations on that structure. A relation may be thought of as a table having a number of rows (called *tuples*) and columns (called *attributes*). 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 concise, yet are remarkably powerful [Ullman82]. One important aspect of some query languages is that they are declarative rather than procedural: they allow the user to specify *what* information is desired, rather than *how* this information is to be derived.

The conceptual design of a database is aided by the *entity-relationship model* [Chen 1976]. In this model relations are classified as *entity* relations or *relationship* relations. Each tuple of an entity relation contains an entity identifier along with attributes describing that entity; an example is the entity relation **Employee** with attributes Name, Department, Salary, and YearsService. Each tuple of a relationship relation contains two or more entity identifiers along with attributes describing that relationship between the entities; an example is the relationship relation **Manages**, with attributes Manager, Subordinate, and YearsUnderManager.

Conventional databases are static, in that they represent the state of an enterprise at a single moment of time. Although their contents continue to change as new information is added, these changes are viewed as modifications to the state, with the old, out-of-date data being deleted from the database. The current contents of the database may be viewed as a snapshot of the enterprise at a particular moment of time.

For relational databases to be relevant to monitoring, there must be a means of recording facts that are true only for a certain period of time. In the database area, attention has recently been focused on precisely this issue. Three types of databases have emerged that encode the notion of time: *rollback* databases, which record the history of database activities, *historical* databases, which record the history of the real world, and *temporal* databases, which incorporate both aspects [Snodgrass 1985, Snodgrass & Ahn 1986]. The historical database is the most appropriate model of the dynamic state of computation. Historical databases require more sophisticated query languages than static databases; TQuel (Temporal QUEry Language) is one that supports historical queries [Snodgrass 1985]. Examples of TQuel queries will be given in a later section, after a new approach to monitoring is presented.

4. A Relational Approach to Monitoring

The central thesis of this paper is that historical databases are an appropriate formalization of the information processed by the monitor. The primary benefits include a simple, consistent structure for the information, the use of powerful declarative query languages, and the availability of a catalogue of optimizations. In this approach, the user is presented with the conceptual view that the dynamic behavior of the monitored system is available as a collection of historical relations, each associated with a sensor in the subject system. In making historical queries on this conceptual database, the user is in fact specifying in a nonprocedural fashion the sensors to be enabled, the analysis to be carried out, and even the graphical presentation of the derived data.

Note that we are *not* proposing to actually represent the data as relations in a database. Instead, we will show that an historical database provides a convenient and powerful *fiction* that

guides the processing but does not constrain the representation. In fact, in most cases the relations will never actually collectively exist as data stored either in main memory or on secondary storage.

Such an approach changes the ordering and the character of the traditional monitoring steps described earlier:

Step 1: Sensor configuration
This step is still performed by the user, except the result is a specification of the data to be collected and the placement of the sensors. Such sensors can be quite flexible; the user is only concerned with specifying the high level properties of the sensor. Conceptually, each sensor declared in this manner defines an historical relation available for later use in defining other, derived relations. The relations directly associated with sensors are termed ***primitive*** relations, as contrasted with ***derived relations***, which are not associated directly with sensors. The specification of the primitive relations identify the information available to the monitor.

Step 2: Sensor installation
This step occurs automatically: the sensor is produced by the monitor from the specifications. Relevant aspects of the sensor are communicated to the components of the monitor that need to know this information. The sensor code handles all the necessary interaction with the monitor, including enabling and buffering, and may be customized to the task it is to accomplish and the environment in which it is to execute.

Step 3: Analysis specification
In this step, the user provides one or more historical queries, defined on the primitive relations specified above.

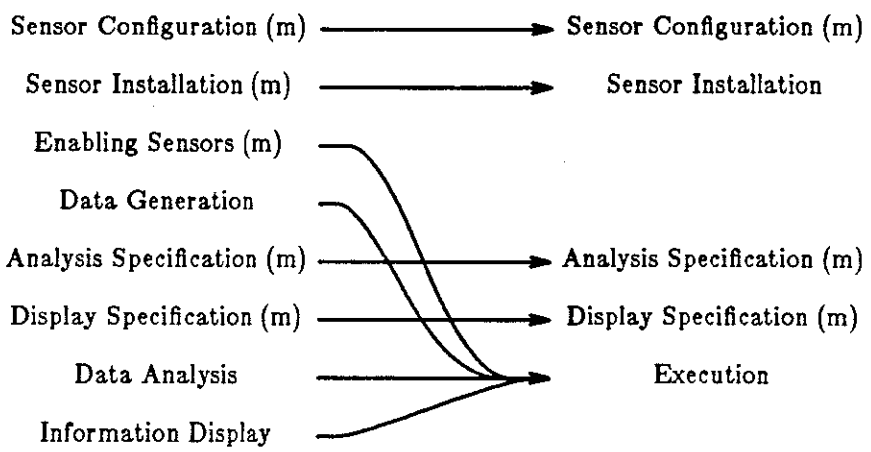
Step 4: Display specification
This step occurs concurrently with analysis specification. By associating entities and relationships with graphical icons (e.g., a square for a processor, a circle for a process, and spatial inclusion (circle within a box) for the relationship "running in"), sophisticated illustrations of dynamic behavior can be generated by the monitor.

Step 5: Execution
This step, comprised of enabling the sensors, generating the data, analyzing the data, and displaying the results, occurs automatically once the queries have been specified. The monitor first analyses the query to determine precisely the sensors that must be enabled to collect the requisite low level information needed to satisfy the query, thereby guaranteeing that extraneous information is not collected. These sensors may be subsequently disabled, and other sensors enabled during the monitoring session based on the data that was collected. Data generation, considered alone, has perhaps the most in common with the traditional monitoring tools. In particular, all the techniques previously developed for data collection are applicable. Data analysis can occur either locally, on the same processing node as the sensor that collected the data, or at a centralized location, or at an intermediate location, depending on the precise query and the capacity of the communication mechanism. The monitor has sufficient information through the sensor specification and the user's query to make the decision as to where the processing will occur. The monitor can also perform optimizations on the query, mapping it into a different query with an identical semantics but improved performance. Information display can also be made more efficient by capitalizing on the fact that only a small portion of the state changes during each transition and by utilizing incremental display algorithms.

The steps for the proposed approach are illustrated in Figure 1, in which the traditional approach is compared with the new approach. The major change is that the sensors are enabled and

the data generated *after* the analysis specification step, allowing the sensors to be enabled automatically based on information from the query. A second change is that some aspects of sensor installation are automated, as described elsewhere [Snodgrass 1986].

Figure 1: Steps of the New Approach to Monitoring



(m): This step is a manual one.

As with the traditional approach, variations are possible. If dynamic sensor installation is supported (say, through the use of breakpoints), this step might be delayed until the execution step. By storing one or more relations in secondary storage, additional iterations of the analysis specification and execution steps (without the enabling and data generation portions) are possible. Finally, defaults supported by the monitor may delay some aspects of some of the steps, (e.g., display specification), until the execution step when they can be performed automatically.

The next four sections discuss this new approach in more detail. Section 5 examines how sensors may be configured by the user. Section 6 deals briefly with how this information is used by the monitor to install the sensors. Section 7 introduces TQuel in a tutorial fashion. The monitoring actions of generating the monitor data and performing the analysis are discussed in Section 8.

This paper will concentrate on the concepts and mechanisms directly affected by the relational approach. The low level data collection mechanism will only be outlined. A future paper will describe this mechanism in detail, focussing on how filtering can be accomplished through an analysis of the user's query [Snodgrass 1986]. The graphical portions of the monitor, involved in the display specification and information display steps, will be dealt with in an even more cursory fashion, and is discussed elsewhere [Shannon 1986].

5. The Sensor Configuration Step

During sensor configuration, the user specifies the data to be collected and the placement of the sensors. Our approach is to provide a simple language for describing what information is to be collected by each sensor, and for indicating where the sensor is to reside. Once such a specification has been processed by the monitor, the code for the sensors will be available to be included in the subject program, the mechanisms will have been set up to get the data packets to the monitor, and the query processing component will know about the primitive relations associated with the sensors defined in the specification. In the implementation described in Section 10, a macro is generated automatically for each sensor; the user inserts an invocation of these macros at appropriate places in the code of the subject system. As with other aspects of the relational approach, complexity has been managed by requiring the user to provide a nonprocedural description of *what* is to be done, leaving the issue of *how* this task is to be done to the monitor, while ensuring that the monitor has sufficient information to make this determination.

In the remainder of this section, we introduce an example subject system and discuss some sensors that might be defined in this system. Since the user is encouraged to think of sensors as defining historical primitive relations, we will employ the entity-relationship model to describe the sensors. As the syntax of the sensor description language is not critical, the sensors will be specified informally, rather than in that language. Although the entity-relationship model can also be used to describe the data collected by hardware monitors, we will ignore this possibility. Details on the sensor description language and on the data collection mechanisms themselves appear in a separate

paper [Snodgrass 1986].

Throughout this paper, the following subject system (an operating system) will be assumed. There are three types of operating system entities known to the monitor: Processor, Process, and Mailbox. In this example, there are several processors, which execute the processes. At any point in time, a process may be executing on only one processor, though processes can execute on more than one processor over their lifetime. A process may send messages to a mailbox, where they will be queued until a process executes the receive operation on the mailbox. If a receive operation is executed on an empty mailbox, the process will block until a message is sent to that mailbox. Several processes may be blocked on a mailbox. Although this example is of necessity oversimplified in comparison with actual operating systems, it should be sufficient for the purposes of this paper. We will now attempt to capture the behavior of this system within the relational model.

Entity relations must be made available for each entity type. The name of each is identical to the name of the type. The **Processor** entity relation contains one attribute, the processor identifier. This relation is always enabled; its associated sensor is placed in the configuration manager which handles the restarting of crashed processors. The **Process** entity relation contains two attributes, the process identifier and the state, one of *Ready* (i.e., the process is scheduled but not currently running), *Running* (the process is currently running on a processor), *Blocked* (the process is waiting on a mailbox), or *Done* (the process has halted or aborted)[2]. This relation is always enabled and is associated with a sensor in the process manager. Finally, the **Mailbox** entity relation contains one attribute, the mailbox identifier, and is always enabled. Its sensor is located in the process communication manager.

Within the monitor, relations are differentiated temporally: there are *event* relations and *interval* relations. Entity relations are always interval relations, for they model entities while they exist in the subject system. Each interval relation contains two implicit attributes, the time the modeled

[2] The State attribute is an enumeration, and hence is not one of the entity types mentioned previously.

interval began, and the time the modeled interval ended[3]. Figure 2 shows the three entity relations, with user names denoting the internal entity identifiers. Most of the entities were created when the system was brought up at 1:00:00 and destroyed when the system was halted at 4:00:00. Interval relations are associated with two sensors, one determining when the interval began and one determining when the interval ended. The first task of the data analysis portion of the monitor is to construct intervals from the data packets generated from these sensors.

[3] The partitioning into explicit and implicit attributes was done for language design reasons; see [Snodgrass 1985] for more details.

Figure 2: Entity Relations

Processor (Processor):

Processor	(From)	(To)
A	1:00:00	4:00:00
B	1:00:00	4:00:00

Process (Process, State):

Process	State	(From)	(To)
P1	Ready	1:00:00	2:00:00
P2	Ready	1:23:24	2:05:12
P1	Running	2:00:00	2:15:37
P2	Running	2:05:12	2:45:29
P1	Ready	2:15:37	2:45:30
P2	Waiting	2:45:30	2:54:20
P1	Running	2:45:30	2:52:47
P1	Done	2:52:47	4:00:00
P2	Ready	2:54:20	2:56:10
P2	Running	2:56:10	2:57:05
P2	Done	2:57:05	4:00:00

Mailbox (Mailbox):

Mailbox	(From)	(To)
M1	1:00:00	4:00:00
M2	1:00:00	4:00:00
M3	1:00:00	4:00:00
M4	1:00:00	4:00:00
M5	1:00:00	4:00:00
M6	1:00:00	4:00:00
M7	1:00:00	4:00:00

Relationship relations can be either event relations or interval relations. A tuple in an event relation describes a change in the state of the system which occurred at a particular instant of time. An example is the **SendMessage** event relation, which has two explicit attributes, a Process and a Mailbox, and one implicit attribute, the time the event occurred (see Figure 3). The tuple (P1, M3, 2:00:05) in this relation represents the instantaneous event of "Process P1 sent a message to Mailbox M3 at time 2:00:05." The content of the message is not recorded in this relation.

Figure 3: An Event Relation

SendMessage (Process, Mailbox):

Process	MailBox	(At)
P1	M3	2:00:05
P1	M4	2:00:06
P1	M7	2:51:13

There are four other relations defined for this system (see Figure 4). The **RunningOn (Process, Processor)** interval relation describes which Process is running on which Processor. Since the system state is constantly changing, the relations evolve over time. For instance, the tuple (P1, B) may be valid in the **RunningOn** relation for only a few milliseconds, and new tuples are added to the **SendMessage** relation as messages are sent. The **Accesses (Process, Mailbox)** interval relation describes which mailboxes a process can send to or receive from, and is always enabled. The **Waiting (Process, Mailbox)** relation lists the processes blocked while waiting to receive from a mailbox. Finally, there is a **Clock** event relation which contains no explicit attributes. The **Clock** relation is treated specially by the monitor; it is generally used to specify sampling, as will be seen below.

Figure 4: Remaining Primitive Relations

RunningOn (Process, Processor):

Process	Processor	(From)	(To)
P1	A	2:00:00	2:15:37
P2	B	2:05:12	2:45:30
P1	B	2:45:30	2:52:47
P2	A	2:56:10	2:57:05

Accesses (Process, MailBox):

Process	MailBox	(From)	(To)
P1	M3	1:00:00	2:57:23
P1	M4	1:00:00	2:57:24
P1	M7	1:00:05	2:57:23
P2	M7	1:23:24	2:40:29

Waiting (Process, MailBox):

Process	MailBox	(From)	(To)
P2	M7	2:45:29	2:54:20

Clock:

(At)
1:00:00
1:00:01
1:00:02
1:00:03

The primitive relations contain timestamps from a gloabl clock maintained across the entire system. Unfortunately, it is theoretically impossible to synchronize imprecise physical clocks over a geographically distributed network with nondeterministic transmission times[Lamport 1978]. However, Lamport does give an algorithm for maintaining a global clock with a bounded imprecision that maintains the invariant that messages are received at a global time that is later than the global time the message was sent. The partial ordering of local events necessary for debugging will be preserved and the (unknown) total ordering will embed this partial ordering. This 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. A second option 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., [Birrell & Nelson 1983]; 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. Given these considerations, we will assume that a global clock is implemented by a distributed algorithm, and is available to each processor. 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.

6. The Sensor Installation Step

In the previous step, the user specified the sensors in a sensor description language. At the same time, the location of the sensor was indicated. The sensor specification is used by the monitor to

- generate the code for each sensor (in the implementation described in Section 10, the code is in the form of a C macro);
- possibly allocate buffers, packet identifiers, counters, and bit vectors for enabling the sensors;
- create primitive relations to be referenced in queries; and
- record information concerning the sensors for later use.

Compilation and linkage of the subject system also occurs in this step. This step is entirely automatic, and generates a fully instrumented subject system. The details of this process appear elsewhere [Snodgrass 1986].

7. The Analysis Specification Step

The sensor configuration provides the information necessary to install the sensors; the historical queries on the primitive relations associated with these sensors provides the information necessary to automate the remaining steps by specifying the content of derived relations. In this way, information not anticipated by the designer of the monitor may still be requested by the user, provided the basic information (i.e., the primitive relations) is available to the monitor. Historical queries are expressed in the temporal query language TQuel [Snodgrass 1985]. TQuel is a general temporal query language, augmenting the (static) relational tuple calculus query language Quel [Held et al. 1975] with additional constructs and providing a more comprehensive semantics by treating time as an integral part of the database. TQuel includes fifteen other statement types, supporting the creation and destruction of databases and relations, storage structure modification, bulk copy of data, and consistency, integrity, and concurrency control. As these statement types are not relevant to the subject of this paper, they will not be discussed further. Instead, we will briefly review the Quel retrieve statement, then present an extended example employing the TQuel retrieve statement.

7.1. The Quel Retrieve Statement

The Quel retrieve statement selects a subset of the tuples in one or more relations, extracts one or more attributes from the tuples in this subset, and combines the attributes into result tuples. The retrieve statement works in conjunction with the range statement. Assume that the two relations mentioned earlier, **Employee** and **Manages**, are available. The statement

```
range of E is Employee
```

specifies that the tuple variable `E` will represent, for example, the tuples of **Employee** on any subsequent retrieve statements.

The retrieve statement creates a new relation whose tuples satisfy a boolean expression. The expressions appearing in the retrieve statement contain constants and attributes from previously defined tuple variables. For example, the following query finds all employees making more than

Ken, who is their manager:

```
range of E2 is Employee
range of M is Manages
retrieve into ToPromote (Name = E.Name)
     where E.Name = M.Subordinate and M.Manager = E2.Name
          and E.Salary > E2.Salary and E2.Name = "Ken"
```

Example 1: Who should be promoted?

This query results in a new relation, called **ToPromote**. The target list "**(Name = E.Name)**" specifies the attribute(s) of the new relation. The *where* clause specifies which tuples will contribute toward the new relation. The retrieve statement thus consists of a *attribute specification* component (the target list) and a *tuple selection* component (the where clause).

7.2. Example TQuel Retrieve Statements

Since TQuel is a superset of Quel, all valid Quel statements are also valid TQuel statements. By utilizing only the target list and where clause in TQuel, many interesting questions may be asked. For instance, to select the processes which are currently Ready, use

```
range of E is Process
retrieve into ReadyToRun (E.Process)
     where E.State = Ready
```

Example 2: Which processes are currently Ready?

ReadyToRun has only one explicit attribute, a Process (see Figure 5). Since the underlying relation (**Process**) was an interval relation, **ReadyToRun** is also an interval relation.

Figure 5: A Derived Interval Relation

ReadyToRun (Process):

Process	(From)	(To)
P1	1:00:00	2:00:00
P2	1:23:24	2:05:12
P1	2:15:37	2:45:30

Intervals can be derived from other intervals. The **WaitingOn** relation identifies those processes which can unblock the currently blocked processes by sending messages:

```
range of W is Waiting
range of A is Accesses
retrieve WaitingOn (Blocked = W.Process, CanUnBlock = A.Process)
     where W.Mailbox = A.Mailbox and E.Process = W.Process and E.State = Blocked
```

Example 3: Which processes can unblock the blocked processes?

This characterization of "CanUnBlock" is conservative since it includes processes which may in fact not be able to unblock another process (if, for instance, they never send messages to the relevant mailbox).

Given **WaitingOn**, the relation **WaitingOnProcessor**, specifying the processors running the processes which have the capacity to unblock the currently blocked processes, may be defined:

```
range of RO is RunningOn
range of WO is WaitingOn
retrieve WaitingOnProcessor (WO.Blocked, CanUnBlockProcessor = RO.Processor)
     where WO.CanUnBlock = RO.Process
```

Example 4: Which processors may potentially unblock processes?

This information is of more than academic interest, since it identifies those processes which may be permanently blocked if a particular processor crashed.

So far, all the example queries were syntactically correct Quel statements, although the semantics is more involved, since the database contains the implicit time attribute. TQuel also includes two additional clauses in the retrieve statement: the valid clause and the when clause. The *valid* clause is similar semantically to the target list. The *when* clause is similar to the where clause.

Recall that the target list specifies the attributes to appear in the derived relation. In TQuel, the target list specifies the *explicit* attributes, and the *implicit* attributes (those containing time values) is specified by an additional clause. The query

```
range of S is SendMessage
retrieve ResumedbyP1 (W.Process)
     valid at end of W
     where S.Mailbox = W.Mailbox and S.Process = P1
     when S overlap W
```

Example 5: Which processes were resumed by process P1?

determines those processes which were initially blocked on a mailbox, then resumed as a side effect of a message being sent by P1 to the mailbox. Since the valid-at clause was used, the resulting relation is an event relation (see Figure 6).

Figure 6: A Derived Event Relation

ResumedbyP1 (Process):

Process	(At)
P2	2:54:20

The valid-at clause can also be used to indicate sampling. The user can request that the **RunningOn** relation be sampled every ten seconds through the query

```
range of C is Clock[10]
retrieve RunningOnEvery10Seconds (RO.all)
     valid at C
```

Example 6: Sample the **RunningOn** relation.

The valid-at clause indicates that the user is interested in the tuples of **RunningOn** only at clock ticks (occurring, in this case, every second).

If the query is defining a derived interval relation, the valid-from-to clause specifies the delimiting instants of the time interval. This clause also takes a variant of path expressions as an argument. To determine the *latency of resumption*, that is, the interval between a message being sent and the recipient being unblocked,

```
retrieve ResumptionLatency (W.Process)
    valid from S to end of W
    where S.Mailbox = W.Mailbox
```

Example 7: When was the Process waiting unnecessarily?

Each tuple in **ResumptionLatency** starts when the message is sent and stops when the process stops waiting (see Figure 7).

Figure 7: Another Derived Relation

ResumptionLatency (Process):

Process	(From)	(To)
P2	2:51:13	2:54:20

Aggregate functions are found in Quel; they have been extended somewhat in TQuel [Gomez & Snodgrass 1986]. These functions refer to groups of tuples rather than individual tuples. For instance, the relation

```
retrieve MBoxQLength (W.MBox, Length = Count(W.Process by W.MBox))
```

Example 8: How long is the queue of waiting processes for each mailbox?

will compute the current length of the queue (of waiting processes) of each mailbox. The contents of this relation change over time, as do all the others. The 'by' clause partitions the tuples according to the MBox attribute; the Count aggregate function is then applied to each partition. The aggregate functions in Quel (and thus TQuel) are **min**, **max**, **avg**, **sum**, and **count**. The total running time, ready time, and blocked time for a process and the the percentage of time a process was Running versus the time it was Ready or Blocked can also be calculated by using aggregate functions.

In converting Quel to TQuel, the syntax was changed as little as possible. The attribute specification component now includes the valid clause, and the tuple selection component now includes the when clause. A few additional aggregate operators complete the syntactic changes to

the language. The TQuel semantics is an extension of the Quel semantics; both are based on the tuple calculus [Snodgrass 1985].

The graphical attributes of the primitive relations may be specified when the sensors are configured or when the queries are specified. Graphical aspects are associated with both entity and relationship relations. There is flexibility in both the iconic representations and the graphic attributes. For example, the shape, color, intensity, size, and position can each be fixed or can be tied to the value of an attribute. Details of the display specification and generation steps are beyond the scope of this paper.

8. The Execution Step

The expressive power of TQuel has a cost: the monitor must be able to determine which sensors to enable, what calculations to perform, and how to display the results, all from the TQuel query. Fortunately, there has been much work on processing relational query languages, and the results of these efforts can be applied in this setting as well. This section will address generating the data and analyzing the data. The relational model also facilitates filtering the data packets and displaying the derived relations; those aspects are beyond the scope of this paper. To provide the context for this discussion, we first review how a conventional database management system (DBMS) processes queries.

8.1. The Relational Algebra

Tuple calculus queries, such as those formulated in Quel, express *what* derived information is desired, letting the DBMS determine *how* the information is to be derived. Relational algebra expressions serve the latter purpose. The DBMS converts each tuple calculus query into an algebraic expression. As this expression is often quite inefficient, optimizations are applied that convert the initial expression into a semantically equivalent one that is more efficient.

In this paper, we will use only a few common relational operations [Ullman 1982]:

Selection

If F is a formula involving constants, attribute names, arithmetic comparison operators and logical operators, then $\sigma_F(R)$ is the set of tuples t in R such that, when the appropriate components of t are substituted for the occurrences of the attribute names in F, then the formula F becomes true. For example,

$$\sigma_{\texttt{E.Dept="Toy"}}(\textbf{Employee}_{\texttt{E}})$$

denotes the set of tuples in **Employee** who work in the Toy department. The subscript on **Employee** indicates that the tuple variable E has been associated with the relation through a range statement.

Projection

If R is a relation with k attributes, we let $\pi_{d_1 d_2 \cdots d_m}(R)$, where d_i is a attribute name, denote the set of m-tuples $a_1 a_2 \cdots a_m$ such that there is some k-tuple $b_1 b_2 \cdots b_k$ in R for which the i^{th} component in a is the d_i^{th} component in b. For example,

$$\pi_{\texttt{E.Name,E.Salary}}(\textbf{Employee}_{\texttt{E}})$$

denotes a relation with two attributes, E.Name and E.Salary.

Cartesian product

Let R and S be relations with k_1 and k_2 attributes, respectively, then $R \times S$, the cartesian product of R and S, is the set of tuples with $k_1 + k_2$ attributes whose first k_1 components form a tuple in R and whose last k_2 components form a tuple in S. For example,

$$\textbf{Employee}_{\texttt{E}} \times \textbf{ToPromote}_{\texttt{T}}$$

denotes a relation with five attributes, E.Name, E.Dept, E.Salary, E.Manager, and E.YearsService from the **Employee** relation and T.Name from the **ToPromote** relation. The non-uniqueness of attribute names is inconvenient; we make the names unique by subscripting the tuple variable to each relation name.

To convert a Quel query into a relational algebra expression, first take the cartesian product of the underlying relations (each associated with a tuple variable used in the query), apply a selection with the formula from the where clause, and then apply a projection, with the attributes from the target list. For example, the query given in Example 1 of Section 7.1 retrieving the **ToPromote** relation has the following algebraic equivalent

$$\textit{(E1)}\quad \pi_{\texttt{E.Name}}(\sigma_{\texttt{E.Name=M.Subordinate}}(\sigma_{\texttt{M.Manager=E2.Name}}(\sigma_{\texttt{E.Salary>E2.Salary}}(\sigma_{\texttt{E2.Name="Ken"}}(\textbf{Employee}_{\texttt{E}} \times \textbf{Employee}_{\texttt{E2}} \times \textbf{Manages}_{\texttt{M}})))))$$

The algebra may be extended to handle TQuel's valid and when clauses, involving the extension of the projection and selection operators, respectively. The projection and selection operators

remain, but only involve the explicit, non-temporal domains. The valid clause is handled by a temporal variant of the projection operator, denoted by a superscript of T. This operator will project out those intervals designated by expressions in the valid clause. The when clause is handled by a temporal variant of the selection operator, also denoted by a superscript of T. The subscript for this operator consists of the temporal predicate specified in the when clause. As an example, the query for **ResumedbyP1**, given in Example 5 of Section 7.2, has the corresponding temporal relational algebra expression,

$$\textit{(E2)}\quad \pi_{\texttt{W.Process}}(\pi^{T}_{\texttt{at end of W}}(\sigma^{T}_{\texttt{S overlap W}}(\sigma_{\texttt{S.Mailbox=W.Mailbox}}(\sigma_{\texttt{S.Process=P1}}(\mathbf{Waiting_W} \times \mathbf{SendMessage_S})))))$$

A more substantial modification is to make the operators *incremental*, so that they operate on streams of tuples, one at a time, possibly generating one or more output tuples whenever an input tuple arrives. The selection and projection operators (both conventional and temporal) are straightforward to extend to operating on streams rather than sets. Each such operator would generate at most one output tuple for each input tuple, and no tuples would have to be stored, assuming that the projection operator does not perform duplicate elimination. The cartesian operator is more complex, for two reasons: it is a binary operator and it requires internal storage. It stores the tuples arriving from the left, and concatenates all of these tuples to tuples arriving from the right, thereby generating multiple output tuples for each input tuple. The brute force cartesian operator requires storage for all the input tuples; more space efficient variants also exist.

Once the relational algebra expressions for the TQuel queries have been generated, they can be used to enable sensors and analyze the incoming data.

8.2. Algebraic Optimization Transformations

The term "optimization" is a misnomer; a more accurate term is "improvement", for an optimal solution almost never results. However, we will continue to use this term, with the understood proviso.

One benefit of using the relational model with monitoring is that traditional optimization techniques may be utilized directly. One example is the transformation

$$\sigma_F (R_1 \times R_2) \rightarrow R_1 \times \sigma_F (R_2)$$

which applies if the predicate F only involves attributes from R_2. This transformation can dramatically reduce the number of tuples generated by the cartesian product, since uninteresting tuples are discarded *before* rather than *after* the cartesian product. This transformation may be applied twice to Expression *(E1)* given in the previous section to obtain

$$\pi_{\texttt{E.Name}}(\sigma_{\texttt{E.Name=M.Subordinate}}(\sigma_{\texttt{M.Manager=E2.Name}}(\sigma_{\texttt{E.Salary>E2.Salary}}(\mathbf{Employee}_{\texttt{E}} \times \sigma_{\texttt{E2.Name="Ken"}}(\mathbf{Employee}_{\texttt{E2}})) \times \mathbf{Manages}_{\texttt{M}})))$$

This optimization can also be applied to the Expression *(E2)*, with the substitutions

- `S.Process=P1` for F.
- **Waiting** for R_1.
- **SendMessage** for R_2.

resulting in

$$\textit{(E3)} \quad \pi_{\texttt{W.Process}}(\pi^{T}_{\texttt{at end of W}}(\sigma^{T}_{\texttt{S overlap W}}(\sigma_{\texttt{S.mailbox=W.mailbox}}(\mathbf{Waiting}_{\texttt{W}} \times \sigma_{\texttt{S.Process=P1}}(\mathbf{SendMessage}_{\texttt{S}})))))$$

A collection of such transformations has been developed for the conventional relational algebra [Smith 1975].

A second class of transformations involves the primitive relations. These transformations improve the algebraic expression by enabling fewer sensors, or by replacing sampling with tracing, or by sampling less frequently. Approximately ten transformations, each with several variants, have been developed thus far. These transformations are discussed in detail elsewhere [Snodgrass 1986].

Finally, a third class of transformations select a more efficient variant of an operation based on the temporal ordering of the input tuples to the operator and the desired temporal ordering of the

output. For example, the cartesian product operator in its most general form must store internally all incoming tuples from the left, so that they can be later concatenated with incoming tuples from the right. If the tuples on both sides were in temporal order, and their overlap was desired, a much more efficient cartesian product may be used:

$$\sigma_{t_1 \text{ overlap } t_2}(E_1 \times E_2) \rightarrow E_1 \times_1 E_2$$

where $\times_1$ denotes the particular variant of the cartesian product. This operator would only store those tuples from the left that could possibly overlap with those from the right, discarding the rest from internal storage. This transformation may be applied to Expression *(E3)* resulting in

(E4) $$\pi_{\texttt{W.Process}}(\pi^{T}_{\texttt{at end of W}}(\sigma_{\texttt{S.mailbox=W.mailbox}}(\mathbf{Waiting}_{W} \times_1 \sigma_{\texttt{S.Process=P1}}(\mathbf{SendMessage}_{S}))))$$

The transformations from the three classes are repeatedly applied (in order) to the algebraic expression until no more are applicable.

8.3. Data Generation

The final result of the optimization phase is an algebraic expression for each query specified by the user. This expression may contain one or more of the following operators: π, σ, π^{T}, σ^{T}, and $\times$. Recall from Section 8.1 that these operators are incremental, in that they take streams of input tuples and possibly generate one or more output tuples whenever an input tuple arrives. The expression is started by having the primitive relations (e.g., **Waiting** in Expression *(E4)*) generate initial tuple streams. The initial tuples flow into the specified operators.

8.4. Data Analysis

Data generation and analysis proceed in parallel. After the algebraic expression is primed with tuple streams from the constants and primitive relations, the tuples flow through the expression in an incremental fashion. One profitable way to view the process is to visualize the parse tree of the expression, with tuples flowing up the arcs. The tuples flowing out of the expression comprise the

historical relation that was specified in the original TQuel query. Performing this analysis in realtime allows the low level data to be discarded after participating in the analysis, with only the derived information stored if desired.

9. Comparison with the Traditional Approach

This paper has argued that monitoring complex systems is fundamentally an information processing activity, and that the relational model provides an effective formalization of this information. In this section, we summarize the steps in the relational approach, then discuss how the new approach addresses problems with applying each step of the traditional approach to monitoring to a complex system.

Step 1: Sensor configuration
Sensors are described in a sensor description language as a collection of primitive event and interval relations. The user also specifies the location of these sensors within the code of the subject system. This description forms the conceptual view that the dynamic behavior of the subject system is available as the collection of historical relations.

Step 2: Sensor installation
The code for the sensors is generated by the monitor. This step is entirely automatic, resulting in a fully instrumented subject system.

Step 3: Analysis specification
TQuel queries are made on this fictional database.

Step 4: Display specification
At the same time, the user specifies the graphical representation of the derived relations.

Step 5: Execution
The queries are first converted into relational algebra expressions, which are optimized through the application of a series of transformations. Processing is started by enabling sensors associated with the primitive relations appearing in the expression. As tuples flow through the expressions, other sensors are enabled, thereby creating other tuple streams. The tuples flowing out of the expressions are displayed as directed by the user.

This approach provides solutions to many of the problems encountered in the application of the traditional approach to monitoring in the presence of complexity.

- *Sensor Configuration*
  One difficulty is communicating the configuration to the monitor, which is distributed along with the sensors. The format of the collected data potentially must be known by all components that handle this data, including the analysis and display components. This issue involves the physical distribution of the monitor. A second difficulty involves the correctness of the sensor code. When monitoring information is used in debugging, the annoying task of debugging the debugger arises. The approach taken by most systems fails to resolve this problem; only a fixed number (10-20) of predefined sensors are usually provided, implying that

future users of the monitor will need only the information determined at the time the monitor was implemented. Such an approach unnecessarily limits the usability of the tool.

In the proposed approach, the relevant aspects of sensors are specified in a high level sensor description language. The translator for this language automatically handles the details of generating the code for each sensor and communicating needed information to the monitor, thereby greatly reducing the chance for error.

- *Sensor Installation*

  The problem here lies in the possible physical distribution of the subject program. In a centralized system, each sensor will reside in an individual program. In a complex system, programs may be physically distributed. Hence the monitor must contend with not knowing until rather late where each sensor resides.

  In the new approach, the sensors are handled automatically by the monitor, freeing the user from being concerned with details of how the sensors are implemented or on which processor each is executing.

- *Enabling Sensors*

  There are two difficulties involved here. One is specifying *which* sensors are to be enabled, a task made difficult by a late binding of program to machine and the sheer magnitude of the number of machines. Determining which processor a process is executing on, out of a large number of processors, can be a tedious and time consuming chore. The task is rendered even more difficult when there are a collection of sensors to be enabled, each on a different subset of processors. The second difficulty is in performing the operation of enabling a remote sensor, while still ensuring that protection between processes is not compromised. Both difficulties involve the physical and logical distribution of the subject program. The approach taken by most systems avoids these problems by permanently enabling all the sensors. In a complex system with many sensors, this approach will quickly overwhelm the processing and communication resources with excess data packets, most of which are not used in the subsequent analyses.

  Our approach makes use of a collection of optimizations to determine precisely which sensors to enable. The monitor uses information from the sensor specification and the algebraic expression to automatically enable only relevant sensors.

- *Data Generation*

  The primary difficulty is in collecting monitoring data from distributed sites. A related issue is the volume of data, and the artifact caused by the collection operation itself. Physical distribution is the culprit here.

  The volume of data collected is reduced considerably through filtering. The appropriate sensors are initially enabled, and can be disabled and other sensors enabled as a side effect of the analysis of previously generated data. Special techniques allow temporal ordering of data packets from multiple buffers. Because sensors are generated by the monitor, there is the opportunity for automatically compensating for the monitoring artifact.

- *Analysis Specification*

  Any attempt to understand the behavior of a (logically) distributed program must focus on the interrelationship of events occurring in different processes. The diversity of interactions precludes the menu or simple command language approach favored by most monitoring systems. Instead, more powerful languages expressing complex patterns are required.

  In our approach, TQuel is used to specify the desired information. TQuel is a high level, nonprocedural language. Since TQuel is an extension of Quel, it is relational complete [Codd 1970]. The when clause can be used to specify temporal relationships between events and intervals occurring in the subject system. The valid clause can be used to specify when the derived

events or intervals are to be valid, as well as suggesting that sampling be done. Aggregate functions provide additional expressive power. This language results in a powerful user interface for querying the monitor concerning the behavior of the system.

- *Display Specification*

  When the user is given any choice at all concerning the display of information, the options are generally limited to canned reports.

  In the relational approach, displays are specified by associating graphical attributes with entities and relationships, for both primitive and derived relations.

- *Data Analysis*

  Monitoring complex systems involves sophisticated data analysis. The centralized, brute-force techniques used by most monitors become inadequate as the subject system becomes more complex.

  A collection of conventional and monitoring specific optimization transformations may be applied to the initial algebraic expression, often resulting in dramatic improvements in execution speed. There is the opportunity for analyzing the collected data in a distributed fashion (see Section 12).

While the above analysis demonstrates the many advantages of the relational approach over traditional monitoring techniques, two substantial issues remain: system complexity and performance.

10. Implementation

In order to assess the practical benefits of the relational approach, we have completed one prototype implementation and have made significant progress towards a second implementation. In this section, we will outline the structure of the prototype and discuss its performance.

The system monitored by the prototype was Cm*, a tightly-coupled multiprocessor composed of 50 DEC LSI-11's and a substantial amount of memory [Fuller et al. 1978, Swan et al. 1977]. Two operating systems were available on Cm*, StarOS [Gehringer & Chansler 1982, Jones et al. 1978, Jones et al. 1979] and Medusa [Ousterhout et al. 1980].

The monitor prototype consisted of two main components: a *remote monitor*, performing those functions requiring close interaction with the user, and a *resident monitor*, performing the functions requiring close interaction with the monitored system. This separation is necessary when monitoring a distributed system, where a resident monitor exists 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 data packets and interacts with the operating system, and the remote monitor analyzes and displays the monitoring data. The prototype ignored the issue of displaying the results graphically, and so the display specification and information display steps were omitted.

The remote monitor ran on a Vax under Berkeley Unix and was itself composed of three modules. The *TQuel compiler* translated the query into an initial algebraic expression. The parse tree for this expression was termed an update network, referring to the tuples flowing across the arcs. The movement of tuples through this network was handled by the *update network interpreter.* The *remote accountant* handled the Ethernet protocol, sending tuples to the interpreter and sending commands to the resident monitor.

Two resident monitors were implemented, one on StarOS called *StarMon*, and one on Medusa called *Medic.* The remote monitor on the Vax communicated with the resident monitor on Cm* over an Ethernet [Metcalfe & Boggs 1975], a high bandwidth (3 MBaud) network.

A minimal monitor was implemented, with all aspects carried far enough to demonstrate feasibility and to investigate efficiency aspects. More specifically, the update network, resident monitors, remote accountant, and TQuel parser and code generator were essentially complete. The TQuel semantic analysis phase was only partially implemented and the optimization phase was designed but never implemented. The graphical display aspect was not addressed at all in the prototype.

Several of the components were instrumented to determine the overall performance of the monitor. The rest of this section will briefly discuss the performance of the sensors, the Ethernet protocol, and the update network interpreter. Details are given elsewhere [Snodgrass 1982].

The efficiency of the data collection mechanism is important, for it determines the monitoring granularity (the level of abstraction at which the monitoring takes place). The mechanism implemented supported strong type checking, multiple type managers, and a high degree of filtering. The sensors required 600-1400 microseconds, depending on the amount of data stored in the data packet. This execution time is equivalent to 85-200 store instructions, or 6 to 14 procedure calls. Hence, the

monitoring grain for this implementation of sensors is larger than a procedure call, but perhaps equal to a procedure that does something interesting, in turn calling other procedures. Given this sensor efficiency, with intelligent filtering reducing the monitoring overhead to 1%, the 50 processors would generate approximately 500 event records per second.

The Ethernet protocol is a variant of the Ethernet File Transfer Protocol (EFTP) [Shoch 1979], simulating a transmission from the remote monitor (the host) to the resident monitor (the slave). The protocol uses checksums, timeouts, and packet retransmission for reliability. Using actual record and packet sizes and observing the transmission rate for the standard EFTP, a maximum transmission rate of 600 event records per second was calculated.

The performance of the update network was measured using a small but relatively complex set of TQuel queries. The initial update network, before optimizations were performed, could process approximately 3 input tuples per second (assuming a dedicated Vax 11/780). Two stages of optimization were performed manually to assess their effect. The first stage applied the transformations discussed in Section 8.2. This step resulted in a speedup of 5, to 15 input tuples per second. The second stage involved substituting the interpreter and general operator algorithms with a Lisp function. Conceptually, the entire algebraic expression was converted into a specialized operator. The Lisp function was then compiled by the FranzLisp compiler into Vax assembly language. The resulting code could process approximately 600 input tuples per second. The four transformations performed in pursuit of reasonable efficiency (TQuel query $\rightarrow$ initial update network $\rightarrow$ optimized update network $\rightarrow$ Lisp $\rightarrow$ assembly language) resulted in an improvement of more than two orders of magnitude.

The general result of these measurements is that, given the monitoring granularity supported by this implementation, the monitor can indeed contend with the number of event records generated by the 50 processors in Cm*. Hence, it is possible to implement a monitor supporting the high level conceptual viewpoint of a dynamic relational database on the system's behavior which can be manipulated by a temporal, non-procedural query language, with sufficient efficiency to monitor a large,

complex, distributed system.

11. Comparison with Other Work

The majority of work in monitoring has concerned the development and application of techniques within the context of the traditional approach. In Section 9 we compared the traditional approach with the relational approach. In this section we examine other research that also addressed inadequacies of the traditional approach.

The basic idea behind the approach espoused here, using historical databases to formalize dynamic information, has been suggested in various guises by others. Ripley organized performance information concerning static program structures (e.g., routines, statements) into a hierarchy and represented hierarchical measurement data as ordered n-tuples, such as (program, routine, statement, primitive operation) [Ripley 1977]. He then suggested applying the relational projection operator on this relation, and implemented a simple system to collect data from Snobol programs and project relevant attributes. This paper is the only one suggesting a relational approach that predates ours.

Garcia-Molina, German, and Kohler went a step further, suggesting that the monitoring relation should be tied to sensors (24 are listed in the paper) rather than to the static program structure, and mentioning that the relational query language Sequel could be used to retrieve information from this relation [Garcia-Molina et al. 1984]. No implementation was attempted.

LeDoux and Parker went a step further still and defined a separate relation for each sensor, presenting 14 predefined relations [LeDoux & Parker 1985]. This database is queried via Prolog. A prototype debugger was implemented using this approach.

One other research project has employed the relational model for monitoring information. The High-level Ada Relational Debugger (HARD) [DiMaio et al 1985] is a component of the Ada Relational Translator, in which *all* data structures are relations [Ceri & Crespi-Reghizzi 1983]. The fact that the dynamic behavior is captured in relations is hidden from the user. Internally the relational

algebra is used to manipulate the information, but externally the user writes Ada tasks to specify the monitoring.

None of these papers proposed using algebraic expressions to specify which sensor to enable, nor using monitoring specific optimizations. A second difference is that the static relational model is employed is all of these papers, with the temporal aspect of the monitored data encoded in an ad hoc manner.

Several researchers have proposed high-level languages for specifying the analysis to be performed by the monitor. As just mentioned, Garcia-Molina et al. suggested using Sequel, LeDoux and Parker used Prolog, and DiMaio, Ceri, and Reghizzi used Ada. In the Interactive Distributed Debugger (IDD), Harter, Heimbigner, and King used interval logic [Schwartz et al. 1983], an extension of linear time temporal logic [Lamport 1980], to specify assertions that are tested in real time [Harter et al. 1985]. It is not clear how this logic compares with TQuel in expressive power, or how hard it will be to implement the assertion checker, as an operational semantics has not yet been developed for interval logic.

Bates and Wileden has defined an Event Definition Language (EDL) in order to obtain a *behavioral abstraction*, in which the system is viewed in terms of higher level events, which are defined in terms of primitive events [Bates & Wileden 1983]. EDL is based on regular expressions augmented with a shuffle operator. It can be shown that TQuel is as expressive as EDL. Implementation experiences with EDL have not yet been reported.

Bruegge and Hibbard applied path expressions [Habermann 1975], originally used to specify constraints on parallel computation, to the specification of event sequences [Bruegge & Hibbard 1983]. Actions may be performed when a particular event sequence is recognized. A prototype was implemented on the Accent operating system. Path expressions were also the basis for the expressions used in the when and valid clauses in TQuel, so the languages are similar in expressive power.

While a moderate amount of research has concerned monitoring distributed systems (e.g., [Miller 1985, Model 1978, Nutt 1979]), no one until now has dealt with the issues of sensor

specification, filtering, or tailoring the display of derived relations. We argued in Section 9 that sensor specification in a complex system was a difficult talk to perform manually. Most systems support a fixed collection of predefined sensors, which makes sensor specification trivial, yet tremendously limits the data that can be collected. We also argued that powerful filtering techniques were absolutely vital in limiting the number of generated data packets. Most systems permanently enable all sensors, or force each sensor to be enabled manually. We disagree strongly with LeBlanc and Robbins, who assert that every event must be stored for later analysis for debugging distributed programs [LeBlanc and Robbins 1985]. This requirement is unncecessary restrictive when many (say, hundreds) of sensors are present, and is usually impossible to satisfy in terms of computing and storage resources in a complex system.

12. Future Work

While the anticipated benefits of a relational approach to monitoring have been demonstrated, there are several areas where further work is needed. On the theoretical side, we are developing a formalization of the incremental temporal algebra discussed in Section 8.1. Such a formalization will be used to

- ensure that the operators are well defined;
- prove that the mapping from TQuel to the relational algebra is correct, using TQuel's tuple calculus semantics [Snodgrass 1985];
- prove that the optimizations do not alter the semantics of the expression they are transforming;
- and perhaps suggest further optimizations.

Another area to be investigated is distributing the analysis. In monitoring a distributed system, the analysis generally occurs at a central node, with the data packets sent to this node from buffers in the processors where the sensors were located that generated the packets. However, much of the analysis should occur locally, with only that analysis requiring more global information being performed remotely. One possibility involves the concept from distributed databases of ***horizontal fragmentation***, where a relation is broken into two or more subsets of tuples, the union of which is the original relation [Ceri & Pelagatti 1984]. In distributed databases, each subset may be stored on a separate node. In the monitoring domain, each primitive relation can be fragmented on the

attribute that specifies where the data packet is generated. The algebraic equivalents of queries on such relations may be duplicated for execution locally on each processor, with the resulting tuples sent to the central node, thereby reducing the load on the network. Optimizations that are not applicable at the central node may still apply to the expression when executed separately on the nodes producing the fragments. Exactly how and when this should be done is under study.

One problem with the relational approach is that the queries must be specified before the data is collected or processed. Because this constraint is placed on the ordering of the steps, the relevant sensors can be enabled automatically. Ideally, there should be some way for the user to indicate with arbitrary precision the data to be collected. In this way, the monitor could support activity at any point along the spectrum between traditional monitoring at one end of the spectrum, where the data is first collected and then analyzed, and relational monitoring at the other end, where the query is specified before any data is collected.

A second problem is the danger that an innocuous query will require an enormous amount of computation. Because the non-procedural nature of TQuel shelters the user from the complex processing resulting from the query, the user has less intuition concerning the cost of queries. Tools need to be developed that indicate the expense of evaluating queries.

Finally, there are implementation issues that should be studied. The goals of the prototype implementation were to demonstrate the feasibility of the relational approach and to identify potential problems requiring further investigation. It was successful in both aspects. However, the prototype lacked optimization and graphic display components and was baroque and inefficient. We are working on a second implementation that will include these components. While the prototype demonstrated the feasibility of the relational approach, we hope to show with the second implementation that a robust, reliable, efficient monitor based on this approach can be constructed.

13. Acknowledgements

I wish to thank William Wulf, Anita Jones, Joseph Newcomer, and Zary Segall for valuable comments and suggestions on all aspects of this research, and M. Satyanaranan and K. Schwan for detailed comments on this paper. In the prototype implementation, Peter Highnam helped with the design of the EtherNet and implemented Medic, and Ivor Durham implemented the first version of

the StarMon sensors. The research performed at Carnegie-Mellon University was sponsored in part by the Defense Advanced Projects Agency (DOD), ARPA Order 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551, the Ballistic Missile Defense Advanced Technological Center under Contract DASG60-81-0077, and through a National Science Foundation graduate fellowship. The research performed at the University of North Carolina at Chapel Hill was supported by the National Science Foundation under Grant No. DCR-8402339, and by an IBM Faculty Development Award.

14. Bibliography

[Agajaman 1975] Agajaman, A.H. *A Bibliography on System Performance Evaluation. Computer,* 8, No. 11, Nov. 1975, pp. 63-74.

[Bates & Wileden 1983] Bates, P. and J.C. Wileden. *An Approach to High-Level Debugging of Distributed Systems.* In *Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on High-Level Debugging,* Ed. M.S. Johnson. Association for Computing Machinery. Pacific Grove, CA: acm, Aug. 1983 pp. 107-111.

[Birrell & Nelson 1983] Birrell, A.D. and B.J. Nelson. *Implementing Remote Procedure Calls.* In *Proceedings of the ACM Symposium on Operating System Principles,* Association for Computing Machinery. Bretton Woods, NH: acm, Oct. 1983 pp. 3.

[Bowie & Linders 1978] Bowie, W.S. and J.G. Linders. *A Software Trace Facility for OS/MVT. Software--Practice and Experience,* 9 (1978) pp. 535-545.

[Bruegge & Hibbard 1983] Bruegge, B. and P. Hibbard. *Generalized Path Expressions: A High Level Debugging Mechanism.* In *Proceedings of the ACM Sigsoft/Sigplan Software Engineering Symposium on High-Level Debugging,* Ed. M.S. Johnson. Association for Computing Machinery. Pacific Grove, CA: acm, Aug. 1983 pp. 34-44.

[Ceri & Crespi-Reghizzi 1983] Ceri, S. and S. Crespi-Reghizzi. *Relational Data Bases In The Design of Program Construction Systems. ACM Sigsoft Software Engineering Notes,* 8, No. 3, July 1983, pp. 17-29.

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

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

[Codd 1970] Codd, E.F. *A Relational Model of Data for Large Shared Data Bank. Communications of the Association of Computing Machinery,* 13, No. 6, June 1970, pp. 377-387.

[Cooperman et al. 1972] Cooperman, J.A., H.W. Lynch and W.H. Tetzlaff. *SPG: An Effective Use of Performance and Usage Data. Computer,* 5, No. 5, sept/oct 1972, pp. 20-23.

[DEC 1983] DEC *Observer: Software Product Description.* 1983. (Unpublished paper.)

[DiMaio et al 1985] DiMaio, A., S. Ceri and C. Reghizzi. *Execution Monitoring and Debugging Tool for Ada Using Relational Algebra.* In *Proceedings of the Ada International Conference on Ada in Use,* Ed. J.G.P. Barnes and G.A. Fisher, Jr. ACM. Paris: Cambridge University Press, May 1985 pp. 109-123.

[Fuller et al. 1978] Fuller, S., J. Ousterhout, L. Raskin, S. Rubinfeld, P. Sindhu and R. Swan. *Multimicroprocessors: An overview and working example. Proceedings of the IEEE,* 66, No. 2, Feb. 1978, pp. 216-28.

[Garcia-Molina et al. 1984] Garcia-Molina, H, Jr., F Germano and W.H. Kohler. *Debugging a Distributed Computing System. IEEE Transactions on Software Engineering,* SE-10, No. 2, Mar. 1984, pp. 210-219.

[Gehringer & Chansler 1982] Gehringer, E.F. and R.J., Jr. Chansler. *StarOS User and System Structure Manual.* Technical Report. Computer Science Department, Carnegie-Mellon University. July 1982.

[Gomez & Snodgrass 1986] Gomez, S. and R. Snodgrass. *A Formal Semantics for Aggregates in TQuel.* 1986. (in preparation.)

[Graham et al. 1982] Graham, S. L., P. B. Kessler and M. K. McKusick. *gprof: a Call Graph Execution Profiler.* In *Proceedings of the SIGPlan '82 Symposium on Compiler Construction,* ACM. Boston, MA: June 1982 pp. 120-126.

[Habermann 1975] Habermann, A.N. *Path Expressions.* Technical Report. Computer Science Department, Carnegie-Mellon University. June 1975.

[Harter et al. 1985] Harter, Jr. P.K., D.M. Heimbigner and R. King. *Idd: An Interactive Distributed Debugger.* In *Proceedings of the Fifth International Conference on Distributed Computing Systems,* May 1985 pp. 1-9.

[Held et al. 1975] Held, G.D., M. Stonebraker and E. Wong. *INGRES--A relational data base management system. Proceedings of the 1975 National Computer Conference,* 44 (1975) pp. 409-416.

[Hoare 1974] Hoare, C.A.R. *Monitors: An Operating System Structuring Concept. Communications of the Association of Computing Machinery,* 17, No. 10, Oct. 1974, pp. 549-557.

[Houghton 1982] Houghton, Jr. R.C. *Software Development Tools.* Technical Report 500-88. U.S. Department of Commerce. Mar. 1982.

[IBM 1984] IBM *VM/370 Real Time Monitor, Program Description/Operations Manual.* IBM Corporation, Cary, NC, 1984.

[Jones et al. 1978] Jones, A.K., R.J., Jr. Chansler, I. Durham, P. Feiler, D. Scelza, K. Schwans and S.R. Vegdahl. *Programming issues raised by a multiprocessor. Proceedings of the IEEE,* 66, No. 2, Feb. 1978, pp. 229-37.

[Jones et al. 1979] Jones, A.K., R.J., Jr. Chansler, I. Durham, K. Schwans and S.R. Vegdahl. *StarOS, a Multiprocess Operating System for the support of Task Forces.* In *Proceedings of the ACM Symposium on Operating System Principles,* Sep. 1979 pp. 117-127.

[Lamport 1978] Lamport, L. *Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the Association of Computing Machinery,* 21, No. 7, July 1978, pp. 558-565.

[Lamport 1980] Lamport, L. *"Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs. Conference Record of the 7th Annual ACM Symposium on the Principles of*

Programming Languages, , Jan. 1980, pp. 174-185.

[LeBlanc and Robbins 1985] LeBlanc, R.J. and A.D. Robbins. *Event-Driven Monitoring of Distributed Programs.* In *Proceedings of the International Conference on Distributed Computing,* IEEE. Austin, TX: 1985 pp. 515-521.

[LeDoux & Parker 1985] LeDoux, C.H. and Jr. D.S. Parker. *Saving Traces for ADA Debugging. SIGAda International Ada Conference,* (1985) pp. 1-12.

[Malone 1983] Malone, J. R. *Implementation of a Retrospective Tracing Facility. Software--Practice and Experience,* 13 (1983) pp. 791-796.

[McDaniel 1982] McDaniel, G. *The Mesa Spy: An Interactive Tool for Performance Debugging.* In *Performance Evaluation Review,* Association for Computing Machinery. Seattle, WA: acm, aug-sep 1982 pp. 68-76.

[Metcalfe & Boggs 1975] Metcalfe, R.M. and D.R. Boggs. *Ethernet: Distributed Packet Switching for Local Computer Networks.* Technical Report CSL-75-7. Xerox Palo Alto Research Center. Nov. 1975.

[Miller 1985] Miller, B.P. . PhD. Diss. ucbcsd, Aug. 1985.

[Model 1978] Model, M. *Monitoring System Behavior in a Complex Computational Environment.* PhD. Diss. Stanford University, Jan. 1978.

[Nutt 1979] Nutt, G. J. *A Survey of Remote Monitors.* Technical Report 500-42. National Bureau of Standards. Jan. 1979.

[Ogle, et al. 1985] Ogle, D., K. Schwan and R. Snodgrass. *The Real-Time Collection and Analysis of Dynamic Information in a Distributed System.* Technical Report OSU-CISRC-TR-85-12. Computer and Information Science Research Center, The Ohio State University. Sep. 1985.

[Ousterhout et al. 1980] Ousterhout, J.K., D.A. Scelza and P.S. Sindhu. *Medusa: an experiment in distributed operating system structure. Communications of the Association of Computing Machinery,* 23, No. 2, Feb. 1980, pp. 92-105.

[Perlis, et al. 1981] Perlis, A., F. Seyward and M. Shaw. *Software Metrics.* Cambridge, MA: MIT Press, 1981.

[Rashid & Robertson 1982] Rashid, R.F. and G.G. Robertson. *Accent: A communication oriented network operating system kernel.* In *Proceedings of the ACM Symposium on Operating System Principles,* ACM. 1982 pp. 64-75.

[Ripley 1977] Ripley, G.D. *Program Perspectives: A Relational Representation of Measurement Data. ieeetse,* SE-3, No. 4, July 1977, pp. 296-300.

[Schwartz et al. 1983] Schwartz, R.L., P.M. Melliar-Smith and F.H. Vogt. *An Interval Logic for Higher-Level Temporal Reasoning.* In *Proceedings of the Second Annual Symposium on Principles of distributed Computing,* Montreal, Quebec: Aug. 1983 pp. 173-186.

[Shannon 1986] Shannon, K.P. *The Display of Temporal Information.* Computer Science Department, University of North Carolina at Chapel Hill, 1986. In preparation..

[Shoch 1979] Shoch, J. *EFTP: A Pup-based Ether file transfer protocol.* 1979. (Unpublished specification.)

[Smith 1975] Smith, D.C. *Pygmalion: A Creative Programming Environment.* Technical Report STAN-CS.75-499. Stanford Computer Science Department. June 1975.

[Snodgrass 1982] Snodgrass, R. *Monitoring Distributed Systems: A Relational Approach.* PhD. Diss. Computer Science Department, Carnegie-Mellon University, Dec. 1982.

[Snodgrass 1985] Snodgrass, R. *A Temporal Query Language.* Technical Report TR85-013. Computer Science Department, University of North Carolina at Chapel Hill. May 1985.

[Snodgrass 1986] Snodgrass, R. *Monitoring Data Collection.* 1986. (In preparation.)

[Snodgrass & Ahn 1986] Snodgrass, R. and I. Ahn. *Temporal Databases. Computer (to appear),* (1986).

[Swan et al. 1977] Swan, R.J., A. Bechtolshem, K.W. Lai and J.K. Ousterhout. *The implementation of the Cm* multi-microprocessor.* In *Proceedings of the National Computer Conference,* AFIPS, 1977 pp. 645-55.

[Tetzlaff 1979] Tetzlaff, W.H. *State Sampling of Interactive VM/370 Users. IBM Systems Journal,* 18, No. 1 (1979) pp. 164-180.

[Tolopka 1981] Tolopka, S. *An Event Trace Monitor For The Vax 11/780.* In *Proceedings of the 1981 ACM Conference,* Association for Computing Machinery. acm, 1981 pp. 121-128.

[Ullman 1982] Ullman, J.D. *Principles of Database Systems, Second Edition.* Potomac, Maryland: Computer Science Press, 1982.

[Wulf et al. 1975] Wulf, W.A., R. Levin and C. Pierson. *Overview of the Hydra Operating System.* In *Proceedings of the ACM Symposium on Operating System Principles,* ACM. Nov. 1975.