TR 86-001

Filtering Monitoring Data

Richard Snodgrass

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

January, 1986

Abstract

Monitoring is an essential part of many program development tools and plays a central role in debugging, optimization, and reconfiguration. One aspect of monitoring, that of collecting only needed information and discarding irrelevant data, termed filtering, is becoming more important. By specifying monitoring in a relational database query language before the data is collected, it is possible to selectively enable only the particular sensors needed. A high degree of filtering results from incorporating a new operator into the algebraic form of the query and transforming the expression into a more efficient one that enables fewer sensors.

Table of Contents

List of Figures

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 debugging. 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. Tuning requires feedback on the program's efficiency, which is determined from measurements on the application while it is running. Finally, monitoring information may also be used internally by the application program. For example, monitoring information is 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. In one study of program development tools [11], a quarter of these tools were highly dependent upon monitoring, including those under the categories of tracing, tuning, timing, and resource allocation. Monitoring is thus an essential function.

Much has been written about monitoring on uniprocessor systems (c.f., the bibliographies [1, 17]) and the general techniques of tracing and sampling are well established. However, one aspect of monitoring, that of collecting only needed information and removing irrelevant dynamic data, termed *filtering*, is becoming more important as programs and their environments become more complex and are distributed to a greater degree. Previous work avoided this issue, either by unrealistically limiting the number of sensors (e.g., [2, 8, 15]), by requiring the user to manually specify the filtering (e.g., [3, 6]), or by accepting the high overhead of permanently enabled sensors as inevitable (e.g., [14]). In this paper, we show that the sophisticated filtering of monitoring data is possible if the sensors and desired analysis are precisely specified before the data collection.

---

[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 [10]. 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 fundamental problem is that traditional monitoring systems do not have sufficient information to automatically perform substantial filtering over a large number of sensors. In Section 2, we present a new approach to monitoring that provides the monitor with this information. The third section discusses a data collection mechanism that is very flexible in terms of the filtering it supports. Section 4 introduces an example that will be used throughout this paper. The next two sections show how the monitor is able to utilize the information provided by the user to perform sophisticated filtering. A final section summarizes the approach and points to further research.

2. Relational Monitoring

The purpose of this section is to provide an overview of a new approach to monitoring, the relational approach, within which sophisticated filtering is possible. This approach utilizes an extension of a conventional relational database to formalize the information processed by the monitor. 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.*

The *relational model* provides both a structuring of the information and manipulations on that structure [5]. 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* [4]. 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. *Historical* databases, which record the history of the real world [25], can model this dynamic computation. Historical databases require more sophisticated query languages than static databases; TQuel (Temporal QUEry Language) is one that includes constructs for historical queries [23, 26]. Examples of TQuel queries will be given after the new approach to monitoring is presented.

We have proposed a new approach to monitoring in which an historical database formalizes the information processed by the monitor [21, 22]. The 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 one or more sensors in the subject system. In making historical queries on this conceptual database, the user is in fact specifying in a non-procedural 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 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.

In this approach monitoring proceeds in five consecutive steps:

Step 1: Sensor configuration
This step results in 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 in Step 1.

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. 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.

Filtering occurs in Step 5, using the specifications provided by the user in Steps 1 and 3. The primary distinctions between the traditional approach to monitoring and the relational approach outlined here are

- In the relational approach, sensor installation procedes automatically from the specification provided in Step 1. Most traditional systems either present a predefined collection of sensors or insist that the user handle all details of sensor installation manually.

- In traditional systems, the analysis is quite constrained; usually the user is given a short menu of predefined analysis options.

- In traditional systems, the data is first collected and stored for later analysis. In the relational approach, the queries are first specified by the user, with the filtering, data analysis, and information display occurring automatically, driven by the queries.

Details of the relational approach, especially how TQuel may be used to specify the analysis and how the analysis proceeds from the queries, is presented elsewhere [22]. Aspects of the display specification and the incremental display of historical relations are under active study [19]. In this paper we will focus on one component of Step 5, enabling the sensors. The next section will examine in detail how this is accomplished.

3. A Sensor Enabling Mechanism

The sensors operate within an environment comprised of a collection of *typed entities*, both passive (i.e., data structures, such as ready queues and semaphores) and active (e.g., processes). Entities have *identifiers*, which are system-dependent names. For instance, in Unix [18], processes are indicated with process-ids; in StarOS [13] entities are named using capabilities, and in Medusa [16] by descriptor-list/offset pairs. Instances of entity types are displayed to the user as character strings; we assume that the operating system supports the mapping between user-oriented character strings and internal entity identifiers. The entity identifiers are assumed to be unique across space and time. Finally, we assume that the monitor can locate an entity given its identifier.

Type managers export operations to be applied to entities of the type(s) supported by the manager; all operations on an entity 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 *target* by the *performer* (the type manager) as a result of a request by an *initiator* (any process). 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 entity type) may have a ReadFile sensor located in the code performing the read operation. Other sensors, such as OpenFile, PhysicalBlockRead, and ModifyProtection, may also be present in the file system. Each sensor is associated with a unique integer, the *sensor identifier*, which is combined with the collected information when it is retrieved by the monitor. The model applies to all levels of granularity; in particular, a type manager and its sensors may be implemented in hardware, firmware, or software.

Sensors may be enabled by setting an *enable flag*. The placement of this flag allows flexibility in the enabling of events. Enable flags associated with a passive entity, such as a file, arbitrate the collection of monitoring information for that entity. Setting the block write event flag associated with a particular file causes information to be collected for file block writes *only for this file* by *any* file system process. On the other hand, setting the file block write enable flag associated with a particular file system process (a portion of the type manager) causes information to be collected for file block writes on *any* file performed *only by this file system process*. The placement of the flags allows filtering along three dimensions, by target, performer, or initiator; the placement of the sensor allows filtering along the fourth dimension: the operation. Each sensor supports filtering in two of this dimensions: the operation and one other dimension. However, several sensors can be associated with an operation (such as the file block write operation in the previous example), each designating a different flag to enable the sensor.

Higher degrees of filtering are also possible. An event may be enabled on a combination of three of the components of the operation, such as a block write operation by *this* file system on *this* file. Filtering on all four aspects represents total control over which event records get generated: a block write operation by *this* file system process on *this* file, as requested by *this* initiator. Achieving higher degrees of filtering requires additional information to be stored and additional processing to

determine if the event is indeed enabled. This extension requires greater than linear space and/or time in the number of entities, and thus is expensive in an environment supporting many entities.

The enable flag can be generalized to an integer counter if multiple enable requests are made by the monitor before the sensor executes. In this case, enabling involves incrementing the counter and disabling involves decrementing the counter.

In the preceding discussion, the assumption was made that the operation is sensed and the information communicated to the rest of the monitor when the operation occurs. Such data packets are called *traced* data packets, since their generation is ***synchronous*** with the operation, and thus with the operation whose target, performer, and initiator is named in the data packet.

Sampled data packets, on the other hand, are generated at the request of the monitor, *asynchronously* with the operation. As an example, a sensor located in the scheduler of an operating system could generate *traced* data packets pertaining to context switching: process x started running at time t_i, process y started running at time t_j, etc. Another sensor located in the scheduler could generate *sampled* data packets at the request of the monitor: process z is now running. A sampled sensor will usually, but not necessarily, clear the enable flag after generating the data packet, thereby causing only one data packet to be generated per request of the monitor.

4. An Example

In this section, we introduce an example subject system (an operating 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. In practice, the user employs a sensor description language to specify these primitive relations.

In this example, 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, an ennumeration having the values *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). 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 interval began, and the time the modeled interval ended. Figure 1 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.

Figure 1: 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 (the initiator) and a Mailbox (the target), and one implicit attribute, the time the event occurred (see Figure 2). 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. This relation is traced on the initiator, meaning that a data packet is constructed if

a message is sent to any mailbox by a process with its associated flag set.

Figure 2: 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 3). The **RunningOn (Process, Processor)** interval relation describes which Process (the target) is running on which Processor (the performer). This relation is sampled on performer; the scheduler of each processor will respond with the current running process when requested by the monitor. 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 **Waiting (Process, Mailbox)** relation lists the processes (the initiators) blocked while waiting to receive from a mailbox (the target) and is traced on the target. Since multiple processes might be waiting on the mailbox, we specify that the enable flag is a counter several bits wide (this option was discussed briefly in Section 3). We also specify that the sensor will decrement this counter each time a data packet is generated; this will permit an important optimization to be discussed later. 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 3: 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

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 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. Historical queries are expressed in the temporal query language *TQuel* [23, 26]. TQuel is a general temporal query language, augmenting the (static) relational tuple calculus query language Quel [9] with additional constructs and providing a more comprehensive semantics by treating time as an integral part of the database.

Here we give two sample TQuel queries. These queries have been chosen to illustrate several important filtering techniques; others based on these primitive relations defined earlier are given elsewhere [22]. The query

```
range of W is Waiting
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 precede end of W
```

Example 1: 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. The act of `P1` sending a message (`S.Process = P1`) causes, directly or indirectly (`S precede end of W`), another process (`W.Process`) that was currently blocked on a mailbox to become unblocked on that same mailbox (`S.Mailbox = W.Mailbox`). This event of unblocking (`valid at end of W`) will be recorded in the event relation **ResumedbyP1** (see Figure 4).

Figure 4: A Derived Event Relation

ResumedbyP1 (Process):

Process	(At)
P2	2:54:20

The valid-at clause can 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 2: Sample the **RunningOn** relation.

`Clock[10]` denotes a clock that ticks once every 10 ticks of the underlying **Clock** relation, which ticks once a second (c.f., Figure 3). The valid-at clause indicates that the user is interested in the tuples of **RunningOn** only at particular clock ticks.

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. To provide the context for this discussion, we first review how conventional database management systems (DBMS) process queries.

5. 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 usually 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 [28]:

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 coming from the where clause, and then apply a projection, with the attributes coming from the target list. The algebra may be extended to handle TQuel's valid and when clauses, involving the extension of the projection and selection operators, respectively. 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. 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.

As an example, the query for **ResumedbyP1** has the corresponding temporal relational algebraic expression

$$(E1)\quad \pi_{\texttt{W.Process}}(\pi^{T}_{\texttt{at end of W}}(\sigma^{T}_{\texttt{S precede end of W}}(\sigma_{\texttt{S.Mailbox=W.Mailbox}}(\sigma_{\texttt{S.Process=P1}}(\mathbf{Waiting}_{W} \times \mathbf{SendMessage}_{S})))))$$

and the query for **RunningOnEvery10Seconds** has the corresponding expression

$$(E2)\quad \pi^{T}_{\texttt{at C}}(\mathbf{RunningOn}_{RO} \times \mathbf{Clock[10]}_{C})$$

6. Incorporating Primitive Relations in the Algebra

Enabling sensors manually in a complex system is very difficult for the user, due to the potentially large number of sensors. One alternative, the brute-force enabling of all sensors, is excessively inefficient. Hence, the monitor should handle the task of determining which sensors to enable, and should only enable the necessary sensors, thereby filtering out unnecessary data packets. Filtering

should occur early and often, so that scarce communication and processing resources are not expended on data which is later discarded.

The monitor must utilize as much information concerning the query to enable the correct sensors. This information is used

- to enable the appropriate traced sensors, and
- to trigger the appropriate sampled sensors at the appropriate times on the appropriate entities.

The strategy employed here was to modify the temporal relational algebra to accommodate primitive relations. The algorithm which translates TQuel statements to algebraic expressions specifies defaults for which sensors to enable. Transformations then map these expressions into semantically equivalent expressions that enable fewer sensors. These transformations are similar to those used to optimize conventional algebraic expressions generated from static query languages.

6.1. The α Operator

To incorporate primitive relations in the algebra, a new operator, α, is used. One parameter of this operator results in a relation indicating which sensors to enable; the output of the operator consists of the tuples generated by these sensors. The subscript of this operator denotes the tuple variable associated with primitive relation, and thus, indirectly, the primitive relation itself. The superscript denotes the strategy employed to collect the data associated with the primitive relation:

- $T{:}P$ Traced, with the enable flag in the performer.
- $T{:}I$ Traced, with the enable flag in the initiator.
- $T{:}T$ Traced, with the enable flag in the target entity.
- $S{:}P$ Sampled, with the enable flag in the performer.
- $S{:}T$ Sampled, with the enabled flag in the target entity.
- $D{:}P$ Traced, then disabled, with the enable flag in the performer.
- $D{:}I$ Traced, then disabled, with the enable flag in the initiator.
- $D{:}T$ Traced, then disabled, with the enable flag in the performer.

$S{:}I$ is not useful, since the initiator is always the monitor process in the case of sampled sensors.

The α operator is substituted for the primitive relation(s) appearing in the expression. For example, the **SendMessage** event relation is traced on the initiator. If this relation was referenced in a query through the tuple variable **S**, it would appear in the algebraic expression as

$$\alpha_{S}^{T:I}(\mathcal{P})$$

The "$\mathcal{P}$" is replaced with an algebraic expression computing the processes for which this sensor is to be enabled. Let us suppose that this expression was simply the constant process entity identifier `P1`. Then this operator would cause the enable flag in the process named by `P1` for the SendMessage sensor to be set. When the process named by `P1` executed a SendMessage operation, the sensor would fire and would generate a data packet containing the entity identifier for the process (i.e., `P1`), the entity identifier for the mailbox being sent to, and a timestamp (c.f., Figure 2). This data packet would be converted into a tuple, which would be contained in the relation output by the α operator.

The $\alpha^{T:P}$, $\alpha^{T:I}$, $\alpha^{T:T}$, $\alpha^{D:P}$, $\alpha^{D:I}$, and $\alpha^{D:T}$ operators have one argument, the relation comprised of entities containing the flag to be enabled. The α^{D} operators, termed *disable-traced*, are associated with sensors that immediately disable their enable flag after generating a data packet. The $\alpha^{S:P}$ operator has two arguments: the entity containing the enable flag (the performer) and a specification of *when* to sample. The $\alpha^{S:T}$ operator has three arguments: *what* to enable (the target entity), *who* to request the sampling of (the performer), and *when* to sample. In all cases, the output consists of the tuples in the relevant primitive relation generated as a side effect of tuples entering the α operator.

A few examples will clarify the differences between the types of α operators. We have already examined the **SendMessage** event relation. The **RunningOn** interval relation is sampled on the performer, and would appear in the algebraic expression of a query referencing it through the tuple variable `RO` as

$$\alpha_{\mathrm{RO}}^{S:P}(\mathcal{P}, \mathcal{P})$$

The first "$\mathcal{P}$" would be replaced with an expression computing processes; the second "$\mathcal{P}$" would be replaced with an expression computing events, at which times the request to sample would be conveyed to the processes comprising the first argument. Generally the secondary argument is a **Clock**

relation. The **Waiting** interval relation is disable traced on the target mailbox:

$$\alpha_{W}^{D:T}(P)$$

When entities arrive from the expression replacing the "P", the Waiting sensor is enabled. However, it is immediately disabled once the operation occurs and the sensor generates the data packet.

The α operator is distinct from the other relational operators in that the output tuples are not simply a function of the input tuples. Instead, the output tuples comprise a subset of a primitive relation, the subset being determined indirectly by the input tuples. Equivalently, the output tuples comprise the data packets generated by the associated sensor, which was enabled as a side effect of input tuples entering the α operator. The incrementation execution of a temporal relational algebraic expression is coupled with the sensors in the subject system through the α operator(s) appearing in the expression.

There is one additional connection between the input and output tuples of an α operator. As an example, we will use $\alpha^{T:P}$. The set of entity identifiers present in the input tuples of this operator will be a superset of the set of entity identifiers present in the **Performer** position of the output tuples, since only those entities were ever enabled. Similar statements can be made of each type of α operator. This connection will soon prove quite valuable.

6.2. Entity Sources

The algorithm translating TQuel queries to algebraic expressions must specify defaults for which sensors to enable, i.e., what the arguments to the α operators must consist of. Here the entity relations (defined in Section 4) for the entity types in the subject system are used. These relations generate tuples naming all existing entities of that type, and are termed *entity sources*. Entity sources are denoted by the entity name; **Process** denotes the entity relation and hence the associated sensor that generates all existing process entities (recall from Section 3 that this sensor may be found in the process manager).

Entity sources complete the terms replacing the primitive relations. The term replacing $\mathbf{SendMessage}_S$ in *(E1)* is

$$\alpha_S^{T:I}(\pi_{\mathtt{Process}}(\mathbf{Process}))$$

Note that a projection operator is necessary for those entity relations that contain more than one attribute. For the second argument of sampled α operators, which specifies when to sample, one of the primitive clock relations is used as a default. The term for $\mathbf{RunningOn}_{RO}$ is

$$\alpha_{RO}^{S:P}(\mathbf{Processor}, \mathbf{Clock})$$

(note that **Clock** is an entity source) and the term replacing $\mathbf{Waiting}_W$ in *(E2)* is simply

$$\alpha_W^{D:T}(\mathbf{Mailbox})$$

The final algebraic expression for the **ResumedbyP1** query can now be presented (compare with *(E2)*):

$$\textit{(E3)}\quad \pi_{\mathtt{W.Process}}(\pi^{T}_{\mathtt{at\ end\ of\ W}}(\sigma^{T}_{\mathtt{S\ precede\ end\ of\ W}}(\sigma_{\mathtt{S.Mailbox=W.Mailbox}}(\sigma_{\mathtt{S.Process=P1}}(\alpha_W^{D:T}(\mathbf{Mailbox}) \times \alpha_S^{T:I}(\pi_{\mathtt{Process}}(\mathbf{Process}))))))$$

as can the expression for the **RunningOnEvery10Second** query (compare with *(E2)*):

$$\textit{(E4)}\quad \pi^{T}_{\mathtt{at\ C}}(\alpha_{RO}^{S:P}(\mathbf{Processor}, \mathbf{Clock}) \times \mathbf{Clock}[10]_C)$$

Entity sources are associated with sensors that are permanently enabled. Note that an entity relation need not be an entity source, if it never appears as a default parameter of an α operator, but an entity source must be an entity relation.

6.3. Data Generation and Analysis

Recall from Section 5 that the temporal relational 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 constants and entity sources (e.g., **Mailbox** and

Process in Expression *(E3))* generate initial tuple streams. These streams are comprised of unary tuples, each containing one entity identifier. Entity identifiers may be accessible by the monitor, or they may have to be supplied by the user. Expression *(E3)* is "primed" with two streams, one containing a tuple for each processor, acquired from system configuration tables, and one containing a tuple for each mailbox, acquired from the process communication manager.

The initial tuples flow into the specified operators. In the case of α operators, the tuples indicate which entities contain the appropriate enable flags to set. The monitor deduces the entity's location from the entity identifier (the mechanism presented in Section 3 assumed that this was possible) within the tuple, and sets the enable flag in the entity, thereby enabling the sensor. Once enabled, the sensors generate data packets, which are gathered and sent to the monitor, where they are separated by sensor identifier. The sensor identifier names a particular α operator associated with a tuple variable ranging over the primitive relation associated with a sensor. The data packets containing the correct sensor identifier form the tuples output by the α operator. Hence, tuples flowing into the α operator indirectly enable various sensors, which generate data packets that eventually comprise the output of the α operator. The tuples flowing into α operators representing intervals specify both when to enable a sensor on a particular entity and when to disable that sensor on that entity. Data generation occurs until all the sensors are disabled in the course of execution.

The processing resulting from Expression *(E3)* is very inefficient. The **Mailbox** sensor generates all the Mailboxes and the **Process** sensor generates all the current processes. The **Waiting** sensor is enabled for the mailboxes and the **Send** sensor is enabled for the processes. The cartesian product generates a tuple for each combination of tuples generated by the **Waiting** and **Send** sensors; the number of generated tuples grows as the product of the total number of block and send operations by all processes. Almost all the tuples are subsequently discarded by the three selection operators. Finally, one explicit and one implicit attribute are projected out, forming the resulting tuples. Similarly, the processing of Expression *(E4)* results in samples taken every second, then concatenated with a tuple for every clock tick, then discarded if the time the sample was taken does not

correspond to the time a particular clock tick occurred. The (in)efficiency of Expressions *(E2)* and *(E4)* is a direct result both of the expressive power of the non-procedural query language TQuel and of the simplicity of the initial translation into a relational algebraic expression. Clearly, this inefficiency is unacceptable and must be corrected if the relational approach is to be a viable one.

6.4. Algebraic Transformations

Using entity sources as arguments to α corresponds to enabling everything. However, transformations may be applied to map expressions into semantically equivalent expressions that enable fewer sensors by replacing entity sources with more constrained expressions that produce fewer entities to enable. One benefit of using the relational model with monitoring is that traditional transformation techniques may be utilized. An example is the transformation

(O1) $$\sigma_F (R_1 \times R_2) \rightarrow R_1 \times \sigma_F (R_2)$$

which is correct if the predicate F does not include attributes from R_1. 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 optimization can be applied to the Expression *(E9)*, with the substitutions

- `S.Process=P1` for F.
- $\alpha_{\texttt{W}}^{D:T}(\textbf{Mailbox})$ for R_1.
- $\alpha_{\texttt{S}}^{T:I}(\pi_{\texttt{Process}}(\textbf{Process}))$ for R_2.

resulting in

(E5) $$\pi_{\texttt{W.Process}}(\pi^{T}_{\texttt{at end of W}}(\sigma^{T}_{\texttt{S precede end of W}}(\sigma_{\texttt{S.mailbox=W.mailbox}}(\alpha_{\texttt{W}}^{D:T}(\textbf{Mailbox}) \times \sigma_{\texttt{S.Process=P1}}(\alpha_{\texttt{S}}^{T:I}(\pi_{\texttt{Process}}(\textbf{Process}))))))))$$

A collection of such transformations has been developed for the conventional relational algebra [20]; these transformations apply directly to the temporal relational algebra.

A second class of transformations involves the α operator. 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; only a few will be discussed here. The first shares some features with the one just given:

(O2) $$\sigma_{t.initiator=K}(\alpha_t^{T:I}(E)) \rightarrow \alpha_t^{T:I}(\sigma_{1=K}(E))$$

In these transformation schemas, variables to be substituted are in italics. Intutively, this transformation states that, rather than enabling a sensor on a large number of processes and then discarding many of the data packets so generated, enable the sensor on only the relevant processes. The reason that $\sigma_{1=K}(E)$ appears on the right hand side rather than simply the constant K is that $\alpha_t^{T:I}$ should be enabled for process K only if E contains K.

In this transformation, E is an arbitrary algebraic expression that returns a relation with one attribute of type process; t is a tuple variable associated with a primitive relation traced on the initiator; K is a constant denoting a particular process identifier; and "1" is the name of the first attribute. In the expression before the transformation is applied, the appropriate sensors is enabled for *all* processes, with most of the resulting data packets (tuples) discarded by the selection operator. In the expression resulting from the transformation, if E contains the process K, then the appropriate sensor *in that process* is enabled. There is no need to discard any data packets, because all the data packets are guaranteed to have a performer of K. This transformation can be applied to Expression *(E5)*, with the substitutions

- S for *t*.
- S.Process for *t.initiator*.
- P1 for K.
- $\pi_{\texttt{Process}}(\textbf{Process})$ for E.

resulting in

(E6) $$\pi_{\texttt{W.Process}}(\pi^{T}_{\texttt{at and of W}}(\sigma^{T}_{\texttt{S precede and of W}}(\sigma_{\texttt{S.Mailbox=W.Mailbox}}(\alpha_{\texttt{W}}^{D:T}(\textbf{Mailbox}) \times \alpha_{\texttt{S}}^{T:I}(\sigma_{\texttt{1=P1}}(\pi_{\texttt{Process}}(\textbf{Process}))))))$$

There is another transformation that is even closer (semantically, not syntactically) to the traditional one:

$$(O3)\quad \sigma^{T}_{\textbf{start of } E_2 \textbf{ precede end of } t}(\sigma_{A=t.target}(\alpha_t^{D:T}(E_1) \times E_2) \rightarrow \alpha_t^{D:T}(E_1 \cap \pi_A(E_2))$$

In the left hand side of this transformation, the attribute A, which must be in E_2, is being used to select tuples generated by $\alpha_t^{D:T}$. An example may be found in Expression *(E6)*, where `S.Mailbox` is used to select tuples generated by $\alpha_W^{D:T}$. However, since the associated sensor is disable traced on the appropriate attribute (the target) anyway, the filtering may occur when enabling the sensor, rather than later, after the unnecessary data packet has been generated. On the right hand side of the transformation, $\alpha_t^{D:T}$ is enabled on only the relevant entities. The temporal selector σ^{T} on the left hand side is necessary because E_2 cannot be used to enable $\alpha_t^{D:T}$ if t finishes before E_2.

There are three restrictions on the application of this transformation:

(1) Only attributes associated with the tuple variable t may be used by operators on the tuples produced by the expression; those found in E_2 are not available for further use.

(2) A must be an attribute in E_2.

(3) The attribute **start of** t is not needed.

This transformation may be used on Expression *(E6)*, using the substitutions

- `W` for t.
- `W.Mailbox` for *t.target*.
- **Mailbox** for E_1.
- $\alpha_S^{T:I}(\sigma_{1=P1}(\pi_{Process}(\textbf{Process})))$ for E_2.
- `S.Mailbox` for A.
- "`S` **precede end of** `W`" for "**start of** E_2 **precede end of** t", since they are equivalent (`S` is associated with an event relation, and `W` with an interval relation).

resulting in

$$(E7)\quad \pi_{W.Process}(\pi^{T}_{\textbf{at end of } W}(\alpha_W^{D:T}(\textbf{Mailbox} \cap \pi_{S.Mailbox}(\alpha_S^{T:I}(\sigma_{1=P1}(\pi_{Process}(\textbf{Process})))))))$$

There are many variants on this transformation, each on different kinds of α operators and each having different constraints.

It should be noted that complications arise when multiple α operators referring to the same primitive relation are present in a collection of queries. Either this situation must not be allowed, or the monitor must be able to sort out the incoming data packets from the sensors and determine which α operators to send each packet to, or the α operators must perform this selection themselves. The correct filtering is still performed at the sensors, in any case.

A third class of transformations involves entity sources. We give two here:

(O4) $\sigma_{1=K}(\mathcal{E}) \rightarrow K$

(O5) $\mathcal{E} \cap E \rightarrow E$

Both derive from the definition of the entity source $\mathcal{E}$ and elementary set theory, and assume that K and E are of the same entity type as $\mathcal{E}$. The first states that selecting a particular entity out of an entity set results in that entity. The second states that taking the intersection of a subset of entities and an entity set result in that subset. The first transformation can be applied to Expression *(E7)*, substituting "P1" for "K" and "$\pi_{\texttt{Process}}(\textbf{Process})$" for "$\mathcal{E}$" to get

(E8) $\pi_{\texttt{W.Process}}(\pi^{T}_{\texttt{at end of W}}(\alpha^{D:T}_{\texttt{W}}(\textbf{Mailbox} \cap \pi_{\texttt{S.Mailbox}}(\alpha^{T:I}_{\texttt{S}}(\texttt{P1})))))$

The second transformation can be applied to this expression substituting "$\pi_{\texttt{S.Mailbox}}(\alpha^{T:I}_{\texttt{S}}(\texttt{P1}))$" for "$E$" and **Mailbox** for "$\mathcal{E}$" to get

(E9) $\pi_{\texttt{W.Process}}(\pi^{T}_{\texttt{at end of W}}(\alpha^{D:T}_{\texttt{W}}(\pi_{\texttt{S.Mailbox}}(\alpha^{T:I}_{\texttt{S}}(\texttt{P1})))))$

The fourth class of transformations involves the **Clock** event relation. This relation can be used to specify sampling rates; several of the transformations allow the monitor to handle this. The transformation

(O6) $\pi_{\texttt{at } C}(\alpha^{S:P}_{t}(E_1, E_2) \times \textbf{Clock}[i]_C) \rightarrow \alpha^{S:P}_{t}(E_1, E_2 \cap \textbf{Clock}[i]_C)$

modifies the second argument of the α operator, which specifies the sampling frequency. The subscript C on **Clock**$[i]$ indicates the tuple variable associated with this predefined relation. Recall that the postfix "$[i]$" denotes a clock that ticks once every i ticks of the underlying clock.

A second transformation

(O7) $\mathbf{Clock}[n] \cap \mathbf{Clock}[n*i] \rightarrow \mathbf{Clock}[n*i]$

allows longer frequences to be used. Both transformations may be applied to Expression *(E4)*. The first transformation results in

(E10) $\alpha_{RO}^{S:P}(\mathbf{Processor}, \mathbf{Clock} \cap \mathbf{Clock}[10]_C)$

and the second in

(E11) $\alpha_{RO}^{S:P}(\mathbf{Processor}, \mathbf{Clock}[10]_C)$

(**Clock** is equivalent to **Clock[1]**) with the result that the **RunningOn** sensor is sampled every 10 seconds, rather than the default sampling frequency.

The transformations from the four classes (traditional, involving the α operator, involving entity sources, and involving the **Clock** event relation) are repeatedly applied in order to the algebraic expression until no more are applicable. A comparison of the processing resulting from the "before" Expression *(E3)* with the processing resulting from the "after" Expression *(E9)* for the **ResumedbyP1** query indicates the increase in efficiency that is possible. The previous examination of Expression *(E3)* revealed that it was very inefficient. The transformed expression, on the other hand, is quite efficient. First, the **Send** sensor is enabled only for the `P1` process ($\alpha_S^{T:I}(\mathtt{P1})$). When `P1` actually sends a message, the mailbox identifier is extracted from the data packet ($\pi_{\mathtt{S.Mailbox}}$) and the **Waiting** sensor is enabled for this mailbox ($\alpha_W^{D:T}$). Since the enable flag is actually an integer (c.f., Section 4), multiple send operations by `P1` are handled correctly. When a process is unblocked, receiving the message, the **Waiting** sensor sends a data packet containing the process identifier and the mailbox identifier. This sensor is also disabled by the monitor, awaiting reenabling when `P1` sends another message to the mailbox. The process identifier and end time are projected out of the data packet, forming the resulting tuple. Using the sample data from Section 4, Expression *(E9)* is primed with the single process identifier for `P1`; tuples flow out of $\alpha_S^{T:I}$ operator (those shown in Figure 2) and into the projection operator, resulting in the tuples (M3, 2:00:05), (M4,

2:00:06), and (M7, 2:51:13). These tuples successively flow into $\alpha_{W}^{D:T}$, which subsequently generates the tuple (P2, M7, 2;45;29, 2:54:20), which flows into the two projection operators, resulting in the tuple (P2, 2:54:20) as shown in Figure 4. The number of generated tuples is linear in the number of send operations by P1.

Expression *(E11)* is also much more efficient than the initial attempt (Expression *(E4)*). The optimized expression specifies that the RunningOn sensor is to be sampled by the scheduler of each processor every 10 clock ticks.

The *relative* increase in efficiency is primarily an indication of the gross inefficiency of the unoptimized expression; the *absolute* efficiency suggests that the optimizations are able to enable the minimal sensors and perform just the computations needed to derive the desired information.

7. Conclusions and Future Work

In the traditional approach, automatic filtering is very difficult, for two related reasons. The major reason is that filtering is encountered too early. There is no way for the monitor to base filtering decisions on how the collected data will be analyzed or displayed; those tasks are specified later. Secondly, even if this information was available, it is not clear how it would be used by the monitor. The filtering capabilities of conventional sensors are primitive, generally consisting of either enabling or disableing the sensor. The analysis specification is also simplistic, thereby reducing the potential benefit gained by analyzing these specifications for hints on filtering.

In this paper we have shown that sophisticated filtering is possible if the sensors and the monitoring queries are precisely specified before the data collection. In the relational approach to monitoring, sensors are specified as historical relations. Filtering is accomplished by associating the enable flag with either the initiator, performer, or target of the operation. Monitoring queries are specified on this relations in TQuel, a non-procedural historical query language. These queries are converted into expression in an incremental temporal relational algebra. The primitive relations are incorporated through a new algebraic operator, α. Optimizations transform the initial expression into a more efficient one that enables fewer sensors, thereby effecting a high degree of filtering. An example

demonstrated the improvement possible on two queries.

Our approach has several advantages over the traditional one:

- *Sensor Configuration and Installation*
  In the traditional approach, the user had to keep track of which programs contained each sensor and where these programs were located in the distributed system. The new approach associates sensors and their enable flags with entities based on a simple model of the environment. For many sensors, the enable flag is in the initiator or the target, decoupled from the performer where the sensor resides. The user need not be concerned with details of where sensors or enable flags are located.
- *Enabling Sensors*
  It was originally the user's responsibility to determine which sensors to enable and to locate these sensors in the distributed system. Our approach makes use of the α operator and a collection of optimizations to determine precisely which sensors to enable. The entity identifier is used to locate the entities which contain the enable flags.
- *Data Generation*
  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.

While sophisticated filtering within the context of the relational approach to monitoring has 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 6. 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 [23]
- prove that the optimizations do not alter the semantics of the expression they are transforming
- perhaps suggest further optimizations

We also want to incorporate data collection techniques other than sampling and tracing into the α operator and its formalization, thereby precisely specifying these data collection techniques.

On the practical side, the next step is to implement a relational monitor to assess the applicability of the optimizations on actual monitoring queries and to determine the effectiveness of the filtering. We have implemented the data collection mechanism discussed in Section 3, as well as the portions of the monitor involved in processing the TQuel queries [22, 24]. Data collection has been implemented on Cm* [7, 27] under the StarOS [12, 13] and Medusa [16] operating systems, and on the Vax under Unix. That both operating systems on Cm* are object-based allowed the implementa-

tions to closely match the conceptual model presented in Section 3; the Unix implementation is not as general nor as consistent with the model. However, we have found it convenient to think in terms of the model even if the implementation does not mirror it exactly. The next step is to implement the optimizations and to make the system available to others for feedback.

Another area still to be investigated is filtering within hardware sensors. While the α operator conceptually could be coupled just as easily to hardware as to software, the actual mechanisms necessary to do so have not been developed.

Finally, it might be desirable to have the monitor play a greater part in sensor specification (e.g., allow it to substitute sampling for tracing to lower the data collection overhead) and sensor installation (e.g., allow it to install the sensors at execution time by using conventional breakpointing mechanisms). How this may be done is an open question.

8. 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 Mahadev Satyanaranan and Karsten Schwan for detailed comments on previous incarnations of this paper. In the prototype implementation, Peter Highnam implemented the data collection mechanism on Medusa and Ivor Durham implemented the first version of the data collection mechanism on StarOS at Carnegie-Mellon University. In the second implementation, still in progress at the University of North Carolina, Chapel Hill, David Doerner, Stephen Duncan, Frederick Fisher, Earle MacHardy, and Steven Reuman all participated in the implementation of the data collection mechanism on Unix. 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, Chapel Hill was supported in part by the National Science Foundation under Grant No. DCR-8402339, and by an IBM Faculty Development Award.

9. Bibliography

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

[2] Bauer, M.J. and J.W. McCredie. *AMS: A Software Monitor For Performance Evaluation And System Control.* In *Proceedings of the Annual Sigmetrics Symposium on Measurement and Evaluation,* Palo Alto, CA: Feb. 1973 pp. 147-160.

[3] Blake, R. *XRAY: Instrumentation for Multiple Computers.* In *Symposium on Computer Performance Modeling, Measurement, and Evaluation,* In proc. of acm. Toronto, Canada: acm, May 1980 pp. 11-25.

[4] 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.

[5] 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.

[6] DEC *SPM: Software Product Description.* 1984. (Unpublished paper.)

[7] Fuller, S.H. *Multi-micro processors: An Overview and Working Example. Proceedings of the IEEE,* 66, No. 2 (1978) pp. 216-228.

[8] 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.

[9] 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.

[10] 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.

[11] Houghton, R.C., Jr. *Software Development Tools.* Special Publication 500-88. National Bureau of Standards. Mar. 1982.

[12] 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.

[13] 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.

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

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

[16] 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.

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

[18] Ritchie, D.M. and K. Thompson. *The Unix Time-Sharing System. Communications of the Association of Computing Machinery,* 17, No. 7 July 1974 pp. 365-375.

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

[20] Smith, J.M. and P.Y-J. Chang. *Optimizing the Performance of a Relational Algebra Database Interface. Communications of the Association of Computing Machinery,* 18, No. 10 Oct. 1975 pp. 568-579.

[21] Snodgrass, R. *A Relational Approach to Monitoring.* In *Proceedings of the Symposium on Practical Software Development Environments,* Pittsburgh, PA: Apr. 1984.

[22] Snodgrass, R. *A Relational Approach to Monitoring Complex Systems.* Technical Report TR85-035. Computer Science Department, University of North Carolina at Chapel Hill. Dec. 1985.

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

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

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

[26] Snodgrass, R. *The Temporal Query Language TQuel.* In *Proceedings of the Third ACM SIGAct-SIGMOD Symposium on Principles of Database Systems,* Waterloo, Ontario, Canada: Apr. 1984 pp. 204-212.

[27] 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.

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