# MONITORING IN A SOFTWARE DEVELOPMENT ENVIRONMENT: A RELATIONAL APPROACH

*Richard Snodgrass*

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

*Abstract:*

*A monitor is an important component of a software development environment. The information collected and processed by a monitor is vital for debugging and tuning programs, and is useful to compilers performing selective optimization. A relational database, extended to incorporate time, is introduced as an appropriate representation of dynamic information concerning a program's execution.* TQuel, *a language permitting high level queries about a program's behavior, is briefly described. Measurements of an initial implementation of the relational monitor show that it can efficiently support the conceptual viewpoint of a dynamic database of a program's behavior.*

## 1. Introduction

This paper presents a new approach to the specification and implementation of monitoring actions. *Monitoring* is the extraction of dynamic information concerning a computational process, as that process executes. This definition encompasses aspects of measurement, observation, and testing.* One use of monitoring is to facilitate the debugging of complex programs [Model 1978]. Monitoring is a necessary first step in understanding a complex computational process, for it provides an indication of *what* happened, thus serving as a prerequisite to ascertaining *why* it happened. Monitoring is also necessary to make efficient use of limited computing resources. In order to selectively optimize a program, the compiler must be able to access information on the execution of the program. A monitor is thus a vital tool in an software development environment, and the information the monitor collects is a useful component of a program's representation.

A uniform yet comprehensive representation of dynamic information regarding a program's progress is essential if this information is to be used by programmers and by tools in a programming environment. This paper introduces one elegant representation, that of a temporal database, and describes an implemented system that allows users to ask high level questions about the behavior of their programs. This approach exploits the significant theoretical and practical results of research in the area of relational databases. By emphasizing the information processing aspects of monitoring, it is possible to formalize the specification of monitoring activity, and to bring powerful techniques to bear to reduce the computational demands of this activity.

## 2. Approach

In an abstract sense, the process of monitoring is concerned with retrieving information and presenting this information in a derived form to the

* 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.

Portions of this research were performed at Carnegie-Mellon University, and was supported in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1551, in part by the Ballistic Missile Defense Advanced Technological Center under Contract DASG60-81-0077, and in part by an NSF fellowship.

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, modeling a particular relationship between collections of entities, may be thought of as a table having a number of rows (called *tuples*) and columns (called *domains*). 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 in their expressiveness [Ullman 1982]. 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 central thesis of this paper is that the relational model is an appropriate formalization of the information processed by the monitor. The user is presented with the conceptual viewpoint that the dynamic behavior of the monitored program is available as a collection of temporal relations. Modelling the dynamic behavior in this fashion has several important ramifications. Queries expressed in existing query languages can be used to select and summarize this information. The relational algebra, suitably extended for temporal relations, provides a convenient executable form for such queries. Optimization strategies result in efficient implementations. In particular, the overhead of data collection, usually a large component, is dramatically reduced by these strategies.

A similar approach was taken by Powell and Linton over the entire software development process [Powell and Linton 1982], and, in particular, in the domain of debugging [Powell & Linton 1983]. In this system, all program information, including the parse tree, symbol table, version history, configuration descriptions, etc., are stored in a relational database system. This uniform representation allows, among other things, queries that refer to entities in the program being monitored. However, their system is based on a static relational database in which temporal information is not stored, a complete query language was never developed, and the system was never implemented. The system described here is more limited in scope, focusing on monitoring. However, this emphasis results in a more thorough design and implementation.

Ceri and Crespi-Beghizzi are more ambitious: *all* data structures in their compiler-interpreter are relations [Ceri & Crespi-Reghezzi 1983]. This approach has severe performance implications. In the system described in this paper the advantages of the the relational model are preserved while avoiding the performance penalties associated with a straightforward implementation using relations.

This paper will describe how the relational view is supported by the monitor. Section 3 discusses how the low level dynamic state of a program may be captured in a collection of relations. Section 4 illustrates how the monitor supports the derivation of more useful, higher level information on the behavior of the program. Section 5 shows how the computational aspects of the monitor can be organized, in particular, how the dynamic incremental updating of temporal relations can be implemented effectively. Section 6 gives an overview of an implementation of the relational monitor, and section 7 discusses of the performance of the monitor. Further details on all of these issues may be found in [Snodgrass 1982].

## 3. Capturing Dynamic Behavior in the Relational Model

The dynamic behavior is recorded by a collection of *sensors* which are placed in the user's program. Each sensor is a section of code which transfers to the monitor information concerning an event or state within the program. The sensor may be inserted by the programmer into the source code, by the compiler into the object code, or by the runtime system into the executing program using the conventional technique of breakpoints. If the sensor is *traced*, then a data record is transferred to the monitor each time a particular event occurs. If the sensor is *sampled*, then a data record is transferred each time the monitor requests the sensor to do so. The data record contains some system dependent information, such as the identity of the sensor, usually a timestamp, as well as information specified by the user. Sensors may be disabled, when the data records they produce are no longer desired, and re-enabled at a later time.

The information collected by the sensors is viewed by the user as a set of relations. These relations are differentiated temporally: there are *event* relations and *interval* relations. A tuple in an event relation describes a change in the state of the program which occurred at a particular instant of time. An example is the **Call** event relation, which has two explicit domains, the calling procedure and the called procedure, and one implicit domain, the time the event occurred; Figure 1 illustrates a portion of the **Call** temporal relation for a monitored compiler. The tuple (DoTypeDec, InsertSymbol, 130) in this relation represents the instantaneous event of "the procedure DoTypeDec called the procedure InsertSymbol at time 130". The time indicates the number of microseconds since the program began execution.

**Call** (Caller, Callee):

| Caller | Callee | (time) |
|---|---|---|
| Main | DoTypeDec | 15 |
| DoTypeDec | InsertSymbol | 130 |
| DoTypeDec | InsertSymbol | 426 |

*Figure 1. An Event Relation*

A tuple in an interval relation specifies a relationship valid during an interval of time. An example is the **Executing** relation, with the single explicit Procedure domain (see Figure 2). The tuple (InsertSymbol, 130, 158) in this relation represents the relationship of "the procedure InsertSymbol executed from time 130 to time 158."

**Executing** (Procedure):

| Procedure | (start) | (stop) |
|---|---|---|
| DoTypeDec | 15 | 923 |
| InsertSymbol | 130 | 158 |
| InsertSymbol | 426 | 466 |

*Figure 2. An Interval Relation*

Since the program state is constantly changing, the relations evolve over time. For instance, the tuple (InsertSymbol, 130, 158) is valid in the **Executing** relation for only a few microseconds, and new tuples are constantly being added.

These relations are called *primitive* relations because they contain information directly accessible to the monitor. For example, the **Call** relation may be maintained either by periodically sampling the program counter, or by tracing the call instruction. Similarly, the **Executing** relation is maintained by sampling the program counter or by tracing both the call and return instructions. Sampling is usually less expensive, but it is also less precise. The primitive relations correspond directly to the sensors that have been inserted into the program. Each time a new sensor is added, a new primitive relation becomes available.

The user is probably not interested in the level of detail of the primitive relations; instead, the user desires more summary information extracted from this detail. A query language provides a powerful mechanism for specifying exactly the information the user wants to retrieve from the program, by specifying the content of *derived* relations. In this way, information not anticipated by the designer of the monitor is still available to the user, provided the basic information (i.e., the primitive relations) is collected by the sensors. This approach is in direct contrast to most monitoring systems, which support a fixed set of analysis commands.

## 4. A Temporal Query Language

TQuel (*T*emporal *QUE*ry *L*anguage) is a high-level, non-procedural monitoring specification language. TQuel is a strict superset of the relational tuple calculus query language Quel [Held et al. 1975], augmenting the retrieve statement with additional constructs and providing a more comprehensive semantics by treating time as an integral part of the database. This semantics has been formalized in the tuple calculus [Snodgrass 1984b], and the syntax and motivation of the language is given in some detail elsewhere [Snodgrass 1984a]; this paper will provide only a few examples to illustrate TQuel's power and flexibility.

To determine when the InsertSymbol procedure was executing,

```
range of E is Executing
retrieve InsertExec
where E.Procedure = InsertSymbol
```

*Example 1. When was InsertSymbol active?*

Since the underlying relation (**Executing**) was an interval relation, **InsertExec** is also an interval relation. The tuple variable E is associated **Executing** relation. The where clause selects tuples in E satisfying the given predicate. This query does not specify any user defined domains; the implicit temporal domains may not be directly manipulated by the user, and are automatically computed by the system (see Figure 3). This query is also a valid Quel query; the semantics are slightly different since the query applies to temporal relations.

**InsertExec**:

| (start) | (stop) |
|---|---|
| 130 | 158 |
| 426 | 466 |

*Figure 3. A Derived Relation*

To determine when the InsertSymbol procedure was active when called, either directly or indirectly by the DoTypeDec procedure,

```
range of I is InsertExec
retrieve DoTypeDec_Insert
where E.Procedure = DoTypeDec
when E overlap I
```

*Example 2. Use of the when clause*

This example illustrates one of the clauses added to Quel: the **when** clause. This clause is the temporal analogue of the where clause. In this case, it selects tuples in the **Executing** and **InsertExec** relations that overlap in time (i.e., that are simultaneously valid at some point). Since InsertSymbol never calls DoTypeRec, the only situation where the intervals overlap is when InsertSymbol was called during an invocation of DoTypeDec.

To select the invocation of InsertSymbol with the longest execution time, when called by the DoTypeDec procedure, either directly or indirectly:

```
range of D is DoTypeDec_Insert
retrieve MaxInsert (ExecutionTime =
         Max (Duration(D)))
```

*Example 3. Use of an aggregate operator*

**MaxInsert** has only one explicit domain (see Figure 4). This example illustrates both a temporal unary operator, Duration, and a conventional aggregate operator, Max, over a temporal relation. Note that the value of the ExecutionTime domain is constant over the intervals [0..130] and [158..454], but is increasing over the intervals [130..158] and [454..466]. As an example, the value of the ExecutionTime domain at time $t = 150$ is 20.

**MaxInsert** (ExecutionTime):

| ExecutionTime | (start) | (stop) |
|---|---|---|
| 0 | 0 | 130 |
| $t$ - 130 | 130 | 158 |
| 38 | 158 | 454 |
| $t$ - 454 | 454 | 466 |

*Figure 4. Another Derived Relation*

## 5. The Update Network

Because TQuel is nonprocedural, queries must be compiled into a representation more amenable to being run. The responsibility of determining what information should be collected (i.e., which sensors are enabled), and what computations should be performed rests with the monitor, not with the user. Both aspects are involved in the generation of an *update network*, which is the target of the TQuel compiler. Static relational database management systems typically convert each calculus-based query into an equivalent expression in the relational algebra [Ullman 1982]. This algebra consists of operators over entire relations. The update network generated from a TQuel query is essentially an executable version of the equivalent algebraic expression, tuned for incremental updating of temporal relations. The nodes in this graph are classified as either *access* or *operator nodes*. Access nodes appear in the tree when the query involves primitive relations. Information in the form of tuples flows out of the access nodes (which are associated with the sensors in the monitored program) and through the network. Operator nodes take tuples from one or more lower nodes and produce tuples which will be sent on to the higher nodes. The entire network is driven by tuples originating in the access nodes.

Update networks support the dynamic incremental updating of temporal relations. Incremental update algorithms for temporal relations accept information in the form of "this relationship between these entities was true for the interval from $t_1$ through $t_2$", and use this information, plus stored information concerning the relation, to derive an updated relation. Primitive relations evolve in time through tuples being added and removed. These changes cause relations derived from a relation to acquire or lose tuples of their own, a process continuing until new information has been completely assimilated by the relations defined in the program. The update network approach emphasizes the flow of information from the sensors through access nodes to the user.

The access and operator nodes present in the update network are instantiated from a set of predefined *generic access* and *generic operator nodes*. Each generic access node is associated with an event type, and thus with the set of sensors generating data records of that type. Access nodes are instantiated from these generic access nodes, and are placed in the network. The generic operator nodes specify various relational operators, such as cartesian product, selection, and projection. Operator nodes are instantiated from these generic nodes by supplying appropriate parameters, then placed in the tree.

The update network is driven by data records originating at the sensors. When a sensor generates a data record, the appropriate tuple is placed on the output arc of all access nodes instantiated from the appropriate generic node. At this point, the tuples start flowing through the network and the processing commences.

## 6. Implementation

The monitor consists 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 program. This separation is necessary when monitoring a distributed program, where a resident monitor exists at each processor, sending collected data to the centralized remote monitor. Functionally, the resident monitor collects the event records and interacts with the program and the operating system, and the remote monitor analyzes and displays the monitoring data.

In the implementation, the remote monitor runs on a Vax under Berkeley Unix [Ritchie & Thompson 1974]. The programs being monitored execute on Cm* [Fuller et al. 1978], a tightly-coupled multiprocessor composed of 50 DEC LSI-11's and a substantial amount of memory. Two resident monitors were implemented, one on StarOS [Gehringer & Chansler 1982, Jones et al. 1978, Jones et al. 1979] and one on Medusa [Ousterhout et al. 1980]. A third resident monitor is currently being implemented directly on the Vax. The remote monitor on the Vax communicates with the resident monitor on Cm* over an Ethernet [ 1975], a high bandwidth (3 MBaud) network.

The query is entered by the user interacting with the front end, which was derived from the Ingres front end, and thus includes the macro and help facilities of the Ingres system [Youssefi et al. 1977]. The TQuel query is compiled into an update network, which is then interpreted. The update network enables the appropriate sensors in the program and the operating system running on Cm*. These sensors generate data records which are collected by the resident monitor and shipped to the Vax over the Ethernet. There they enter the update network, to emerge later as tuples in the requested relation.

An implementation of the monitor is now running, and work is proceeding to develop a more extensive system. More specifically, the update network interpreter, the resident monitors, and the TQuel compiler are essentially complete. The semantic analysis and optimization phases of the TQuel compiler are currently being extended to support further optimization strategies. An update network compiler and several other components not mentioned have been designed, and are currently being implemented.

## 7. Performance Issues

Although the high level conceptual viewpoint of a dynamic relational database on the program's behavior results in a powerful user interface via TQuel, it remains to be shown that a monitor supporting such a viewpoint is sufficiently efficient. Several of the components were instrumented to determine the overall performance of the monitor.

There are many areas where bottlenecks could severely compromise the performance of the monitor. In the following, the primary areas will be examined, focusing on the steps taken to increase performance. This analysis will then be summarized using the common metric of data record rate supported.

One such area is the data collection. Enabling all the sensors and subsequently filtering the data records in the update network resulted in an unacceptably large execution time overhead on Cm*. Furthermore, the other components of the monitor could not contend with this excessive rate. Two solutions were adopted [Snodgrass 1984c]. First, sophisticated filtering techniques were adopted that enabled only the necessary sensors. Sensor enabling occurs as a side effect of the interpretation of the update network. Second, the sensors themselves were generated automatically from a high level description provided by the user. The resulting code was carefully tailored based on these specifications, and made extensive use of existing microcoded operations.

Another potential bottleneck was the Ethernet protocol. The event records are generated by the sensors and placed in temporary storage areas, waiting to be picked up by the resident monitor and sent to the remote monitor via the Ethernet. The protocol [Highnam&Snodgrass 1981] is a variant of the EFTP (Ethernet File Transfer Protocol), simulating a transmission from the remote monitor (the host) to the resident monitor (the slave). This protocol may be thought of as a modified transport protocol using the Pup protocol as the network layer of the communications hierarchy [Boggs et al. 1980, Zimmerman 1980]. The commands are sent in command packets and the data records are placed in the acknowledgement packets. The protocol uses checksums, timeouts, and packet retransmission for reliability. Since the resident monitor is a slave in the protocol, it must wait for the remote monitor to send a packet before it can respond with an acknowledgement containing data. Hence the remote monitor must occasionally send packets even if there are no commands to be sent. The resident monitor indicates in every acknowledgement the amount of buffer space it has free, allowing the remote monitor to adjust the packet transmission rate accordingly. The size and transmission rate of the packets were chosen to optimize performance.

The third area critical to the performance of the monitor is the update network. The TQuel compiler as implemented generates correct update networks, but does not include most of the optimization strategies discussed above. Although the interpreted version was flexible and relatively easy to implement, it had one major drawback: it was slow. The maximum node fire rate was less than 140 per second, corresponding to an input data record rate of less than 7 data records per second. This rate is about two orders of magnitude too low. To achieve such a speedup, it was necessary to abandon some of the flexibility afforded by the interpreter.

The update network compiler, not to be confused with the *TQuel* compiler, translates an update network into a collection of Lisp functions, which are then compiled by the Lisp compiler [Foderaro 1980].

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

Three sets of measurements were taken; one with the update network generated by the existing compiler, one with the update network optimized by hand, using only strategies that could be readily implemented (described elsewhere [Snodgrass 1982]), and one with Lisp functions generated by hand from the optimized update network, again using only strategies that could be readily implemented. It should be emphasized that the measurements only apply to one set of queries, and may not be representative of queries in general. On the other hand, these queries are somewhat complex, involving several tuple variables and temporal clauses.

Since all three update networks were correct, they generated identical output tuples for the same input data records. For a set of 50 test input data records, chosen to produce interesting results, there were 32 output tuples produced.

In the non-transformed network, each input data record resulted in almost 20 node fires. In the optimized update network, each data record resulted in less than 10 node fires. The execution time *per node* went down by 40% from the original to the optimized update network. These two reductions cooperatively increase the number of data records processed per second by a factor of 3.

An even larger increase (a factor of 40) occurs with the update network is compiled. There is only 1 fire per tuple in the compiled version because each update network in this approach is in effect a highly specialized operator node (internal function calls were not counted). In summary, the optimizations on the original update network, coupled with conversion of the update network into Lisp, and then into assembly language, result in the necessary improvement of two orders of magnitude.

At this point, the relative performance can be analyzed. Assuming that filtering reduces the overhead to 1% (detailed measurements of the effectiveness of the various filtering mechanisms have not yet been performed), the 50 processors in Cm* would generate 500 data records per second. Given the observed transmission rates for the standard EFTP, a rough maximum transmission rate is 600 data records per second. Applying update network optimizations and using an update network compiler results in a processing rate of over 600 data records per second on a dedicated Vax 11/780. Thus, the Ethernet and remote monitor can essentially keep up with the 50 processors on Cm*. It is also fair to state that if the situation is changed in some way; say, an additional 20 processors are added to Cm*, or the Ethernet or the Vax is loaded, then the monitor as realized here would *not* be able to sustain an adequate data record processing rate.

## 8. Conclusions

This paper has argued that the relational model provides an effective conceptual model of dynamic information concerning a program's execution. Several specific results were detailed:

(1) A monitoring specification language, TQuel, was developed by syntactically and semantically augmenting an existing query language. This language is the basis for a powerful user interface for querying the monitor concerning the behavior of the program.

(2) Update networks were proposed to implement the dynamic incremental updating of derived temporal relations. The network is composed of access nodes, which interface with the resident monitor, and operator nodes, which carry out the desired computations.

(3) Several general techniques were developed to generate correct and efficient update networks from TQuel queries.

(4) The monitor was implemented. Measurements show that the monitor can essentially keep up with a large multiprocessor.

The point to be emphasized is that it is in fact possible to implement an efficient monitor supporting the high level conceptual viewpoint of a dynamic relational database on the program's behavior which can be manipulated by a temporal, non-procedural query language.

## 9. Acknowledgements

Wm. A. Wulf, Anita Jones, Joseph Newcomer, all of Tartan Labs, and Zary Segall, at CMU, were very helpful in all stages of this research. They provided ideas, encouragement, and software systems on which to implement these ideas.

## 10. Bibliography

[Boggs et al. 1980] Boggs, D.R., J.F. Shoch, E.A. Taft and R.M. and Metcalfe. *Pup: An internetwork architecture. IEEE Transactions on Communications,* COM-28, No. 4, Apr. 1980, pp. 612-24.

[Ceri & Crespi-Reghezzi 1983] Ceri, S. and S. Crespi-Reghizzi. *Relational Data Bases in the Design of Program Construction Systems. SIGPlan Notices,* 18, No. 11, Nov. 1983, pp. 34-44.

[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.

[Foderaro 1980] Foderaro, J.K. *Franz Lisp Manual.* Opus 33b ed. UC Berkeley, 1980.

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

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

[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.

[Highnam&Snodgrass 1981] Highnam, P. T. and R. Snodgrass. *The Cm*/Simon Protocol.* Technical Report. CMUCSD. 1981.

[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.

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

[Jones et al. 1979] Jones, A.K., R.J. Chansler, Jr., 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.

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

[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.

[Powell and Linton 1982] Powell, M. and M. Linton. *The OMEGA Programming System.* 1982. (in preparation.)

[Powell & Linton 1983] Powell, M. L. and M. A. Linton. *A Database Model of Debugging.* In *Proceedings of the SIGSoft/SIGPlan Software Engineering Symposium on High-Level Debugging,* Ed. M. S. Johnson. ACM. Pacific Grove, CA: Mar. 1983 pp. 67-70.

[Ritchie & Thompson 1974] 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.

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

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

[Snodgrass 1984b] Snodgrass, R. *Formal Semantics of a Temporal Query Language.* 1984. (Submitted for publication.)

[Snodgrass 1984c] Snodgrass, R. *Monitoring Data Collection on a Multiprocessor.* 1984. (Submitted for publication.)

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

[Youssefi et al. 1977] Youssefi, K., N. Whyte, M. Ubell, D. Ries, P. Hawthorn, B Epstein, R. Berman and E. Allman. *INGRES Reference Manual.* 6th ed. Electronics Research Lab., University of California, Berkeley, California, 1977.

[Zimmerman 1980] Zimmerman, H, *OSI Reference Model - The ISO Model for Open Systems Interconnection. IEEE Transactions on Communications,* COM-28, No. 4, Apr. 1980.