Application Semantics for Active Monotonic Database Applications

Lane Warshaw, Daniel P. Miranker
Department of Computer Sciences
University of Texas at Austin
{warshaw,miranker}@cs.utexas.edu

Abstract

We refine an active-database application taxonomy, proposed by Stonebraker, to include monotonic log monitoring applications (MLM). MLMs are a subclass of hard rule systems where triggering events are restricted to monotonic relations. We develop a formal semantic model for the MLM class. We then prove the correctness of concurrency schemes for applications within the model. Our results demonstrate that only minimal coupling mode support is necessary for the database integration of hard rule systems obeying the MLM restrictions.

1. INTRODUCTION

Active database technology enhances traditional databases with rules that react to database events. Applications of the technology range from *simple rule systems* (applications with few rules that rarely interact such as integrity constraints) to *hard rule systems* (applications with many rules that significantly interact such as real-time decision control systems) [7,13,21,26].

Active database applications, however, are not merely production systems applied to data within a database; rule computation must obey the ACID properties (Atomicity, Concurrency, Independence, and Durability) of a database. The most widely accepted approach, introduced by the HiPAC project [12], is for active database developers to relate rule processing to database transactions through a pair of *coupling modes*. The modes specify the transaction relationship of 1) database events to condition evaluation and 2) condition evaluation to action execution. This explicit specification of coupling modes by application programmers promises to increase system throughput by maximizing flexibility. However, the progression of research has lead to a proliferation of the number of coupling modes [6,8,12]. As a result, coupling modes often burden application programmers with extremely difficult conceptual specifications. In this paper, we begin deciphering which coupling modes are necessary to achieve useful active database programming.

We have observed that a number of our active database applications that we have developed ranging across point of sale, medical patient, network security monitors, real-time decision control systems, and process control monitors can be classified into a subclass of hard rule systems called *monotonic log monitoring* (MLM) applications [5,11,20,24,25,26][1]. MLMs process real-time data logged to a database. The primary reason a DBMS is chosen is to exploit the database's query and data durability services as a platform for decision-support. A fundamental property of the MLM logs is that they are inserted to, but never updated nor deleted. It is this write-once nature of the

1. It is our conjecture that many other active database programs that are not intrinsically monotone are also mappable to the MLM class.

logs that we exploit to reduce the number of applicable coupling modes. This paper presents a formal study, using active constructions, of the resulting simplifications that can be made of active database programs that obey the MLM restrictions.

1.1. APPROACH

Our starting point is a formal specification of the active database languages presented in [2,18,28]. Section 3 presents the definition of this unified general-purpose active database language. Section 4 expands this language with three increasingly concurrent active execution models. In all three models, rules execution is triggered by an *external event* - an atomic state change to the database performed by a database user or application program.

The first, and most basic execution model, is the *sequential execution model*. This model forms the basis of correctness by reflecting the behavior of active database programs executing as a stand alone application with no other database activity. As such, rules are evaluated sequentially until a *quiescent state* - a state in which no more rules are triggered. Though this model is simple and straightforward, this single user environment is impractical. Further, the simplicity of the model eliminates the need for coupling modes.

The second model presented is the *parallel execution model*. This model expands upon the sequential execution model by allowing concurrent rule execution. Although restricting external event behavior reduces the usefulness of this model, the properties we prove about the parallel model are used as a stepping stone to prove properties about concurrent MLM rule processing.

The most general execution model presented is the *active database execution model*. This model is an unrestricted model in which both external events and rules execute concurrently [18]. As such, the active database execution model accurately portrays modern active database systems executing within a multi-user environment.

Using our three execution models, we present a series of proofs that specify the concurrency schemes for MLM programs. These schemes meet the sufficient conditions for *program correctness*. A program is said to be correct under an execution model iff every possible execution path within the model is equivalent to a path within the sequential execution model (Section 4.3). We divide our analysis into two categories. The first category consists of MLM^+ programs - MLMs that contain only *positive* variables[2]. The second category consists of MLM^- programs - MLMs that contain both positive and *negative* variables[3]. It follows from these definitions that the logical database language Datalog is a proper subset of MLM^- programs [23]. Our proofs exploit the previous results and proof techniques in serializability theory, rule dependency graphs and confluent rules systems [1,9,16,23].

Serializability theory states the conditions upon which concurrent processing is equivalent to a serial interleaving of operations. A well known application of the theory is within database transaction models [15]. In [4], Bernstein states a set of conditions that specify when the order of interfering operations matter (RAW, WAW, and RAW). These conditions are violated when interfering operations are executed in parallel without restrictions. The results of violating the Bernstein conditions is that the database may move to an incorrect state.

2. A positive variable is a database query on the existence of values within a database.
3. A negative variable is a database query that uses the closed world assumption to test for the absence of values within a database.

We exploit serializability theory to describe when the parallel execution of rules interfere with one and another. Rule interference is synonymous to the Bernstien's conditions where if certain conditions are violated, the order of rule execution matters. In this paper, we use Kuo, Miranker and Browne's rule serializability theory based on bipartite rule dependency graphs (Section 5) [14,16].

In Kuo et al, a graph in which a cycle of rules *interfere* with one and another is called a *cycle of dependency*, and the set of rules in a cycle of dependency form a *mutual exclusion set*. Two key theorems presented in [16] describe execution cycles in terms of *cycle serializability*, an execution cycle that is equivalent to some serial execution of rules. These two theorems are 1) the *cycle serializability theorem* which states the parallel execution of all rules in a mutual exclusion set may lead to a non-cycle serializable execution cycle and 2) the *serializability theorem* which states that a parallel execution cycle that does not contain all the rules in a mutual exclusion set is guaranteed to be cycle serializable.

In contrast to serializability theory which describes the properties of interfering rules, confluent rule systems explain the properties of quiescent states [1,9,19,23]. A rule system is confluent when the quiescent state is unique despite rule ordering. Towards this end, we present the program characteristics and concurrency models that are sufficient for MLMs to be confluent.

1.2. RESULTS

Our first result establishes a concurrency scheme for MLM^+ programs. Theorems 2 and 4 prove that a MLM^+ program is correct when all rules in the program are specified in E-C and C-A decoupled modes (Section 6). In fact, MLM^+ programs have been proven to be confluent in the sequential execution model [9]. Theorems 1 and 3 demonstrate that confluence still holds for active database MLM^+ programs with concurrent rules and external events.

Our next findings concern the more general MLM^- programs (Section 7). Concurrency models for MLM^- programs are difficult to assign since they do not contain unique quiescent states [9,23]. Further, active database developers expect external events and the rules that they trigger to appear as atomic state transitions. This assumption necessitates the consideration of the time ordering sequence of external events (Section 7.1). These complications become apparent within applications where it is possible for an incorrect program execution to contain only cycle serializable execution cycles.

Our MLM^- analysis begins with the concurrency scheme for programs executing the parallel execution model (Section 7.2). We exploit Kuo, Miranker, and Browne's work to identify rules that must be stated in E-C and C-A immediate modes. Specifically, Theorem 5 proves that an MLM^- program executing with the parallel execution model is correct when at least one rule in every mutual exclusion set is stated in E-C and C-A immediate modes.

We next analyze MLM^- programs executing with the active database execution model (Section 7.3). We present three decreasingly restrictive concurrency schemes. All schemes exploit the interactions of external event *closures* - the set of all rules that may become active due to the execution of an external event. Graphically, the external event closure is all rules reachable by a depth first traversal in the rule dependency graph rooted by the external event.

The first concurrency scheme for MLM^- programs executing with the active database execution model is overly restrictive. Theorem 6 proves that a MLM^- program is correct when at least one rule in every mutual exclusion set is stated in E-C and C-A immediate modes and all rules in all

external event closures that contain a rule that is connected with a negative edge in the dependency graph are stated in E-C and C-A immediate modes.

The second concurrency scheme for MLM$^-$ programs executing the active database execution model improves concurrency based on transaction characteristics. Our definition of external events are that they are atomic and committed. Theorem 7 exploits this definition by proving that a MLM$^-$ program is correct when at least one rule in every mutual exclusion set is stated in E-C and C-A immediate modes, and all rules in all external event closures that contain a rule that is connected with a negative edge in the dependency graph are stated in E-C and C-A *deferred* modes or stronger. It is important to note that deferred coupling mode semantics allow for rule execution to continue in parallel.

Our third and most general concurrency scheme for MLM$^-$ programs executing the active database execution model further improves concurrency based *external event interference* - the situation in which the parallel execution of the closure of rules triggered by two or more external events may violate *external event sequencing*. Lemma 5 establishes dependency graph regions where external event interference may occur. Theorem 8 proves that a MLM$^-$ program is correct when at least one rule in every mutual exclusion set is stated in E-C and C-A immediate modes, and all rules in all external event closures in which external events *interfere* with one and another are stated in E-C and C-A deferred modes or stronger. It is important to note that many MLMs are embedded applications that have a limited number of external events. This last concurrency scheme exploits this property to improve system throughput.

2. BACKGROUND

2.1. Active Database Rules and Coupling Modes

Expert system rules are *Condition-Action rules* (CA rules). CA rules are evaluated on every update to working memory. Within the active database paradigm, such evaluation may be prohibitive since numerous external events may occur in a multi-user database. Thus, active database rules follow the model proposed by HiPAC [12]. This model extends rules to include an *event section* that describes when to evaluate a rule. The resulting rules are called *Event-Condition-Action rules* (ECA-rules).

The relationship between events, rule execution and database transactions has been addressed in a series of *coupling modes* [12,28]. An ECA rule contains two classes of coupling modes. The first class is the *E-C coupling mode* - the transaction relationship between the occurrence of an event and the condition evaluation. The second class is the *C-A coupling mode* - the transaction relationship between the evaluation of the rule's condition and its action's execution.

Many coupling modes have been proposed. The first and most predominately used modes are *immediate*, *deferred*, and *decoupled* [12]. In immediate mode, execution of the pair occurs in the same transaction; in deferred mode, execution of the second part of the pair occurs just prior to transaction completion; and in decoupled mode, execution of the pair occurs in separate transactions. Other coupling modes that have been proposed include the detached causally dependent in either of parallel, sequential or exclusive modes to give abort and commit semantics for decoupled transactions [6].

2.2. Monotonic Log Monitors (MLM)

This paper presents the simplifications that can be discerned from a class of applications called *monotonic log monitors* (MLM). MLMs are a class of real-time active-database applications [17,24]. Such systems process some form of real-time data logged to a central database. The primary reason a DBMS is chosen as a storage method is to exploit the database's decision-support services. We use this write-once nature of logs to restrict the number of useful coupling modes and to simplify the transactional interface elements.

We believe MLMs represent a significant class of active database applications. Examples range across point of sale, medical patient and process monitoring, and intrusion detection systems [26] . In each of these application areas, there is need to move decision support out of batch-mode and/or human-in-the-loop processing towards increasingly timely trigger driven analysis.

MLMs form a subset of the most complex region of a two-axis active-database application taxonomy. This taxonomy is organized by the complexity of applications' rule components. On one axis of this taxonomy, Michael Stonebraker proposed a classification based on the amount of **search** of the applications' rule systems [21]. *Simple rule systems* have few rules with little interaction; *hard rule systems* have many rules with significant interaction. On the other axis, proposed by Lance Obermeyer [17], is a taxonomy based on the size of the **data** being analyzed.

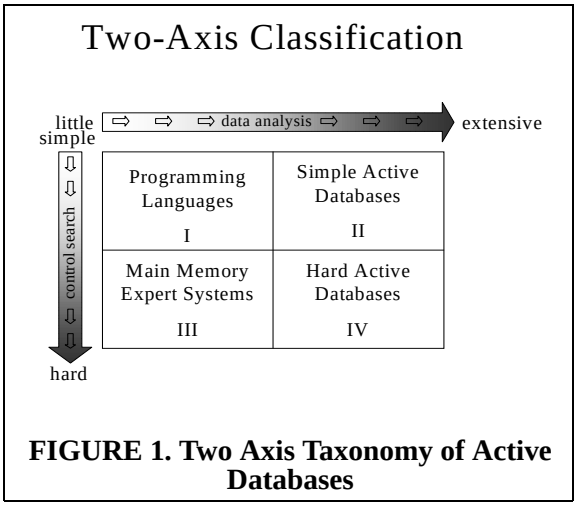

FIGURE 1. Two Axis Taxonomy of Active Databases

This two-axis taxonomy of active database problems yields four distinct regions (Figure 1). Regions I and III are applications that investigate small amounts of data. Such applications do not necessitate the full functionality of a database, and therefore, coupling modes may be omitted.

Regions II and IV, however, are applications that search through large amounts of data. These problems require the data management services of databases, and as such, necessitate coupling modes for transaction processing. Region II consists of "simple" rule systems incorporating applications such as view maintenance, integrity constraint and workflow systems [7,13][4]. Such applications generally treat each rule as a separate program. Consequently, the number of coupling modes for Region II applications is relatively minimal. Region IV consists of "hard" rule systems of which MLMs represent a significant subset. Such applications contain hundreds if not thousands of rules interacting in arbitrary ways. The already complex nature of hard rule systems become unmanageable with the addition of coupling mode semantics [2]. A goal of this paper is to begin deciphering the necessary coupling modes to achieve useful active database programming of Region IV applications.

4. The word "simple" in this context by no means implies that these technologies are trivial.

2.2.1. MLMs, Datalog, and Confluence

MLMs have been shown to have similar characteristics to programs stated in the logical database language Datalog [9]. Datalog programs are rule-based programs with the following properties [23]:

- Rules are *safe* - range restricted.
- Data is monotonic.
- Data is stored in a database.
- Pure Datalog rules are Horne clauses.

The properties of MLM programs obey the first three restrictions.

[1,9,19,23] use these characteristics to establish the sufficient conditions for MLMs to be *confluent*. A rule program is confluent when all eligible serial executions of the program terminate in a unique state regardless of the ordering of rule firings [1]. In particular, MLMs in which no rule conditions contain a negative variable (a test for the absence of facts in a database) have been proven confluent. Further, *stratified* MLMs have also proven confluent [9,19].

This theory is a foundation upon which we build our concurrency models. Yet, it is not all encompassing. First, an underlying assumption of the theory of confluence is that rules are executed atomically and in isolation from other database activity. Active databases, on the other hand, assume an opposite model in which rules execute in parallel with external events and operate according to the semantics of coupling modes.

Secondly, confluence cannot be guaranteed for ECA rule programs in which rules do not monitor for all events (as do expert system rules). However, without loss of generality, we ignore this situation. The justification is that omitting external events can be characterized into either 1) omissions purposefully introduced for efficiency improvements (the developers are not concerned with the undefined behavior that may result), or 2) inadvertent bugs introduced by the active database developer (similar to a semantic bug in a procedural program). In either case, an omission of an event does not represent incorrect behavior introduced by the active database execution model.

2.3. Concurrency Control in Active Database Systems

Concurrency control issues for general purpose active database systems are largely ignored. There are two reasons for this. First, the underlying extensional database can be relied upon to implement the concurrency control features necessary for rule condition evaluation and action execution (especially if the rule's constituents are implemented using the extensional database's query language). Second, most active database systems are designed to address "simple" rule systems. In such systems, program correctness can be ensured if the rules acquire the necessary locks [17,28].

One approach to addressing concurrency control for general purpose active database programs is presented in Correl and Miranker [10]. This scheme attaches isolation specifications to individual rules and collections of rules called modules. Three categories of data isolation are proposed called *guard stability, serializable*, and *exclusive*. Guard stability allows the greatest amount of concurrency, but provides the least amount of isolation from other users. This mode dictates that, at minimum, a tuple accessed during condition evaluation will be available during action execution. Exclusive mode ensures no other transactions will affect the rule system. Serializable mode contains properties in between guard stability mode and exclusive mode.

Though a significant step, a serious deficiency of Correl's method is that it requires the application programmer to determine the system requirements and sensitivity to external state transitions. This paper addresses this deficiency.

3. Definitions

This section presents the active database language definitions used in this paper.

We define a **database table** as an active database relation. A **tuple** is a row in a database table that represents data. The **extensional database**, $\mathcal{E}$, is the non-empty collection of database tables $\{T_0, T_1, \ldots, T_{n-1}\}$.

A **database event** is defined as $V \in \{Insert, Modify, Delete\}$ where *Insert, Modify,* and *Delete* are labels.

Modifications to the database occur using data manipulation commands [2]. A data manipulation command is the pair (*V,T*) where *V* is a database event, and $T \in \mathcal{E}$. The data manipulation commands *a* and *b* are equal iff *a* = (*x,y*) and *b* = (*x',y'*) and $x = x' \wedge y = y'$.

Though usually omitted in this study, data manipulation commands contain data. For example, a database insertion contains an inserted tuple. When necessary, our examples refer to data in the following ways:

(*Insert, T*(*a*)) - Insert tuple *a* into table *T.*
(*Delete, T*(*a*)) - Delete tuples *a* from table *T.*
(*Modify, T*(*a*),*T*(*b*)) - Modify tuples *a* in table *T* to *b*.

We define an active database **rule base**, $\mathcal{R}$, as a non-empty finite set of active database rules. An **active database rule** is the triplet (*E,C,A*) where:

- The *event clause, E,* is a non-empty collection of data manipulation commands, $(V_0, T_0), (V_1, T_1), \ldots, (V_{n-1}, T_{n-1})$, in which a rule monitors for modifications to the database. The execution of any one of the data manipulation commands instigates further processing of the rule.
- The *condition clause, C* , is a condition over some state of $\mathcal{E}$. (To be defined below.)
- The *action clause, A,* is a non-empty sequence of data manipulation commands $(V_0, T_0), (V_1, T_1), \ldots, (V_{n-1}, T_{n-1})$ performed when *C* is *satisfied* in some state of $\mathcal{E}$. (Rule satisfaction is discussed below.)

We define an **active database** as the pair ($\mathcal{E}$,$\mathcal{R}$). Depending on context, we often refer to an active database as an active database program. These terms mean the same thing and are used interchangeably.

For a rule *R* = (*E,C,A*), we sometimes use the notation E^R, C^R, A^R to denote the rule's constituents.

An active database rule's **condition clause** is a relational calculus predicate ranging over the extensional database[5]. Variables within the predicates may be either 1) *positive* or 2) *negated*. Positive variables are existentially quantified variables. Negated variables are identical to classi-

5. Relational calculus predicates are assumed to be safe [23].

cal negation used in Datalog and expert systems languages that use the closed world assumption to test for the absence of values ($\neg\exists$).

Modifications to the database may occur outside of rule execution through an **external event**. We define an external event, $\mathcal{X}$, as a nonempty sequence of data manipulation commands $(V_0, T_0), (V_1, T_1), \ldots, (V_{n-1}, T_{n-1})$ performed atomically at a particular time. External events initiate rule processing. Therefore, with regard to transaction boundaries, we assume that external events are committed. Otherwise, external events may occur in nested subtransactions that can be rolled back. The rolling back of rule execution is beyond the scope of this paper.

Active databases change state over time. Towards this end, we define an **extensional database state**, $\mathcal{D}$, as the state that consists of all the tuples within all the extensional database tables at a particular time. As such, a **table state** is the set of all tuples belonging to a table $T \in \mathcal{E}$ at a particular time. An **active database state** is defined as the pair $(\mathcal{D},\mathcal{T})$ where

$\mathcal{D}$ is a extensional database state.

$\mathcal{T} \subseteq \mathcal{R}$ is the set of triggered rules. (To be discussed below.)

An active database is in a **quiescent state**, $(\mathcal{D}, \varnothing)$, when the set of triggered rules is empty. Two active database states $(\mathcal{D},\mathcal{T})$ and $(\mathcal{D}',\mathcal{T}')$ are equivalent iff all tuples in all table states of $\mathcal{D}$ and $\mathcal{D}'$ are equivalent and $\mathcal{T} = \mathcal{T}'$.

Changes to database state spawn rule evaluation. We say that a rule R **monitors** a table T when $\exists(V, T) \in E^R$. Likewise, we say that a table T is **monitored** if $\exists R \in \mathcal{R}$ such that $\exists(V, T) \in E^R$.

Without loss of generality we make the following assumption:

Assumption: $\forall T \in \mathcal{E}$, $\exists R \in \mathcal{R}$ such that R monitors T.

Our assumption implies that all data manipulation commands within rule actions operate on monitored tables. In practice, actions may contain operations on unmonitored tables and/or outside sources (such as printing to a user interface). Such operations do not effect our study and are henceforth ignored.

3.1. Functions

Following is a list of functions used in this paper.

$C^R(\mathcal{D})$: Where $R \in \mathcal{R}$ and $\mathcal{D}$ is an extensional database state.
$C^R(\mathcal{D})$ = *true* if C^R evaluates to true in state $\mathcal{D}$. In this case, we say that C^R is *satisfied*.
$C^R(\mathcal{D})$ = *false* otherwise.

$A^R(\mathcal{D})$: Where $R \in \mathcal{R}$ and $\mathcal{D}$ is an extensional database state.
$A^R(\mathcal{D}) = \mathcal{D}'$, where $A^R(\mathcal{D})$ executes the sequence of data manipulation commands A^R starting from state $\mathcal{D}$ resulting in a new database state $\mathcal{D}'$.

$T \in Pos(C^R)$: Where $R \in \mathcal{R}$ and $T \in \mathcal{E}$.

$T \in Pos(C^R)$ = *true* if the table T is bound to a positive variable in C^R.

$T \in Pos(C^R)$ = *false* otherwise.

$T \in Neg(C^R)$: Where $R \in \mathcal{R}$ and $T \in \mathcal{E}$.

$T \in Neg(C^R)$ = *true* if the table T is bound to a negated variable in C^R.

$T \in Neg(C^R)$ = *false* otherwise.

Triggers(*d*): Where *d* is a sequence of data manipulation commands $(V_0, T_0), (V_1, T_1), \ldots, (V_{n-1}, T_{n-1})$.

Triggers(*d*) is the set rules $R \in \mathcal{R}$ such that $\exists i, 0 \le i \le n-1, (V_i, T_i) \in E^R$. For purposes of analysis, a data manipulation command that does not create a state change (e.g., inserting a repeated copy of a tuple) does not add its monitoring rule to the result set.

Apply($\mathcal{X}$,$\mathcal{D}$): Where $\mathcal{X}$ is an external event and $\mathcal{D}$ is an extensional database state.

Apply($\mathcal{X}$,$\mathcal{D}$) = $\mathcal{D}'$, where *Apply*($\mathcal{X}$,$\mathcal{D}$) executes $\mathcal{X}$ starting in state $\mathcal{D}$ resulting in a new database state $\mathcal{D}'$.

3.2. Sequence of States

The database moves from an active database state $(\mathcal{D}_n,\mathcal{I}_n)$ to $(\mathcal{D}_{n+1},\mathcal{I}_{n+1})$ in the following ways.

1). A rule $R \in \mathcal{R}$ *links* the states $(\mathcal{D}_n,\mathcal{I}_n)$ to $(\mathcal{D}_{n+1},\mathcal{I}_{n+1})$ iff $R \in \mathcal{I}_n$ and either[6]

i. $C^R(\mathcal{D}_n)$ is *true*, and

ii. $A^R(\mathcal{D}_n) = \mathcal{D}_{n+1}$, and

iii. $(\mathcal{I}_n - R) \cup Triggers(A^R) = \mathcal{I}_{n+1}$

or

i. $C^R(\mathcal{D}_n)$ is *false*, and

ii. $\mathcal{D}_n = \mathcal{D}_{n+1}$, and

iii. $\mathcal{I}_n - R = \mathcal{I}_{n+1}$

2). An external event $\mathcal{X}$ links the states $(\mathcal{D}_n,\mathcal{I}_n)$ to $(\mathcal{D}_{n+1},\mathcal{I}_{n+1})$ iff

i. $Apply(\mathcal{X}, \mathcal{D}_n) = \mathcal{D}_{n+1}$, and

ii. $\mathcal{I}_n \cup Triggers(\mathcal{X}) = \mathcal{I}_{n+1}$

We define an **execution graph** as the graph $G_e = (V, E)$ where the vertices $V \in G_e$ represent active database states and the edges $E \in G_e$ are states that are linked as described above. An active database program's **execution path** is the path through an execution graph taken by a particular execution.

6. Due to the properties of MLMs, we do not have to consider rules that are un-triggered as described in [27].

3.3. Monotonic Log Monitor Definitions

Consider an active database program ($\mathcal{E}$,$\mathcal{R}$) that executes the sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \mathcal{X}_1, \ldots, \mathcal{X}_{n-1}\}$. For $\forall R \in \mathcal{R}$ and $\forall X \in \mathcal{X}$, a table $T \in \mathcal{E}$ is **monotonic** iff

$$\forall (V, T) \in \{A^R \cup X\},\ V = Insert \qquad \text{(EQ 1)}$$

Informally, a table *T* is monotonic iff all rules and all external events perform only insertions into *T*. Note, it is not necessary to know the entire set $\mathcal{X}$ a priori; it is sufficient to constrain $\mathcal{X}$ to contain only insertions to *T*. Notationally, we say:

MonotonicTable(*T*): Where *T* is satisfied by the Equation 1.

For an active database ($\mathcal{E}$,$\mathcal{R}$) and a rule $R \in \mathcal{R}$, *R* is a **monotonic active database rule** iff

$$\forall (V, T) \in A^R,\ V = Insert \qquad \text{(EQ 2)}$$

Informally, a rule *R* is monotonic iff all data manipulation commands in its action are insertions.

Monotonic Log Monitor (MLM) - Consider an active database program ($\mathcal{E}$,$\mathcal{R}$) that executes the sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \mathcal{X}_1, \ldots, \mathcal{X}_{n-1}\}$. ($\mathcal{E}$,$\mathcal{R}$) is a MLM iff:

$$\forall T \in \mathcal{E},\ MonotonicTable(T) \qquad \text{(EQ 3)}$$

Equation 3 implies that all rules in a MLM program are monotonic.

We distinguish two categories of MLMs. The first category, $\mathbf{MLM}^+$, are MLMs containing only positive condition variables. The second category, $\mathbf{MLM}^-$, are MLMs containing both positive and negated condition variables.

4. Active Database Execution

This section formalizes the active database execution models presented in [2,18]. We present three slightly different models that vary depending on their restrictiveness with respect to concurrency. In all three models, rule execution begins with the occurrence of an external event.

The first model, the *Sequential* model, is based on algorithms presented in [2,3,27]. The model execution proceeds by locking the database from external events and serially executing until quiescence.

Though straightforward, the *Sequential* model forfeits concurrency. Therefore, we introduce the the *Parallel* and the *ActiveDatabase* execution models. The *Parallel* model allows for concurrent rule execution but locks the database from external events during rule processing. The general *ActiveDatabase* model, based on an aggregation of the models presented in [2,18], allows for concurrent execution of both external events and rules.

We begin our discussion by introducing the semantics of parallel execution of active database rules and external events.

4.1. Atomicity and Parallel Rule Execution

Section 3.2 presented the linking of active database states as if rules are executed atomically. However, this is not the case. Operations within an extensional database are gauranteed to be

atomic iff the operations live within a transaction. In an active database, the set of atomic operations are expanded to include rule conditions, rule actions and an external events (but not entire rules) since each such operation must be executed within a transaction. This atomicity does not come at the expense of concurrency. The locking mechanisms of the underlying database allow for concurrent execution of transactions. Yet, since entire rules do not necessarily live in a single transaction, parallel rule execution may lead to an incorrect database state.

Coupling modes handle the issue of rule atomicity by allowing the user to force the desired execution sequence. The database locking mechanisms in conjunction with coupling modes result in the following semantics:

1. Conditions in E-C immediate mode are executed in *sequential* nested sibling transactions from the spawning transaction.
2. Conditions in E-C deferred mode are delayed until the end of the spawning transaction and then executed in parallel.
3. Conditions in E-C decoupled mode are executed in independent top transactions.

Statements 1 through 3 are identical for rule actions [12].

The above semantics result in following two concurrency semantics for linking states.

1. Atomic transition. Coupling mode semantics imply that rules stated in E-C and C-A immediate modes are executed atomically (within the same transaction). Therefore, we write,
   $(\mathcal{D}_k,\mathcal{T}_k) \xrightarrow{X} (\mathcal{D}_{k+1},\mathcal{T}_{k+1})$ where X is either an external event or rule stated in E-C and C-A immediate modes, and $(\mathcal{D}_{k+1}, \mathcal{T}_{k+1})$ is the resulting database state.
2. Parallel transition (also called a *parallel execution cycle*). We write,
   $(\mathcal{D}_k,\mathcal{T}_k) \xRightarrow{X} (\mathcal{D}_{k+1},\mathcal{T}_{k+1})$ where $X = \{\mathcal{R}_0, \ldots, \mathcal{R}_x \cup \mathcal{X}_0, \ldots, \mathcal{X}_y\}$, $\mathcal{R}_m \in \mathcal{R}$, $\mathcal{X}_n \in \mathcal{X}$, and $(\mathcal{D}_{k+1}, \mathcal{T}_{k+1})$ is the resulting database state. The algorithm for a parallel transition with the above set X is as follows:

   ```
   while(X ≠ { })
       do_in_parallel
           choose R from X, and remove it from X.
           R links (D_k,T_k) to (D_k',T_k') where R is spawned in the transaction
           model specified by its coupling modes (or executed atomically if R is an
           external event or a rule stated in E-C and C-A immediate modes).
   ```

   Much of the remaining focus of the paper is to clarify the meaning of $(\mathcal{D}_{k+1},\mathcal{T}_{k+1})$.

We are now ready to present our three execution models.

4.2. Execution Models

4.2.1. Sequential execution model

Method Name: *Sequential*
Input: $\mathcal{D}_i$, $\mathcal{X}_j$

Output: $\mathcal{D}_k$ where $k > i$.

Algorithm: For each $\mathcal{X}_j$ executed, the following algorithm is spawned:

0). $(\mathcal{D}_i,\varnothing) \xrightarrow{\mathcal{X}_j} (\mathcal{D}_{i+1},\mathcal{T}_{i+1})$, $i := i + 1$

1) while $\mathcal{T}_i \neq \varnothing$

Begin loop

2). Select $R \in \mathcal{T}_i$

3). $(\mathcal{D}_i,\mathcal{T}_i) \xrightarrow{R} (\mathcal{D}_{i+1},\mathcal{T}_{i+1})$, $i := i + 1$

End loop

4). return $\mathcal{D}_i$

4.2.2. Parallel execution model

Method Name: *Parallel*

Input: $\mathcal{D}_i$, $\mathcal{X}_j$

Output: $\mathcal{D}_k$ where $k > i$.

Algorithm: For each $\mathcal{X}_j$ executed, the following algorithm is spawned:

0). $(\mathcal{D}_i,\varnothing) \xrightarrow{\mathcal{X}_j} (\mathcal{D}_{i+1},\mathcal{T}_{i+1})$, $i := i + 1$

1) while $\mathcal{T}_i \neq \varnothing$

Begin loop

2). Select $R = \mathcal{T}_i$

3). $(\mathcal{D}_i,\mathcal{T}_i) \xRightarrow{R} (\mathcal{D}_{i+1},\mathcal{T}_{i+1})$, $i := i + 1$

End loop

5). return $\mathcal{D}_i$

Step 2 of the *Parallel* model has been the subject of much research [6,22,27]. Many of the first active database languages modeled the *Sequential* algorithm; one rule is selected from T_k on each cycle [27]. *Parallel* improves system throughput by allowing rules to execute concurrently. The *Parallel* model, similar to the REACH rule system [6], selects **all** rules in T_k for concurrent execution on each cycle. This choice may seem overaggressive since if all the rules were really executed in parallel, incorrect behavior could result. However, coupling modes handle the issue of rule atomicity by providing a mechanism to force the necessary execution sequence in cooperation with underlying database's locking mechanisms that manage the detailed data interactions.

4.2.3. ActiveDatabase execution model

Method Name: *ActiveDatabase*

Input: $\mathcal{D}_i$, $\mathcal{X}_j$

Output: $\mathcal{D}_k$ where $k > i$.

Algorithm: Each $\mathcal{X}_j$ executed in a quiescent state spawns the following algorithm:

0). $(\mathcal{D}_i,\varnothing) \xrightarrow{\mathcal{X}_j} (\mathcal{D}_{i+1},\mathcal{T}_{i+1})$, $i := i + 1$

1) while $\mathcal{T}_i \neq \varnothing$

Begin loop

2). Select $R = \mathcal{T}_i$

3). $(\mathcal{D}_i, \mathcal{T}_i) \xrightarrow{\{R \cup \mathcal{X}_{j+m} \cup ... \cup \mathcal{X}_{j+n}\}} (\mathcal{D}_{i+1}, \mathcal{T}_{i+1})$, $i := i + 1$

End loop

5). return $\mathcal{D}_i$

The *ActiveDatabase* execution model selects a set of rules at Step 2 to be concurrently evaluated and executed with external events in Step 3.

Notationally, we refer to an active database program *Y* that executes using the execution model *X* beginning in state $\mathcal{D}_i$ executing a sequence of external events $\mathcal{X}$ and terminating in state $\mathcal{D}_j$ as:

$$X_Y(\mathcal{D}_i, \mathcal{X}) \Rightarrow \mathcal{D}_j \quad \text{(EQ 4)}$$

4.3. Correct Active Database Execution

We define an active database state $\mathcal{D}_j$ as a **sequential database state** iff $\mathcal{D}_j$ is the initial database state $(\mathcal{D}_0,\varnothing)$ or $\mathcal{D}_j$ is a state that is linked from $\mathcal{D}_0$ through the sequential application of data manipulation commands. We note that the initial database state, $(\mathcal{D}_0,\varnothing)$, is a correct active database state: it the state in which no data manipulation commands have occurred.

Corollary 1. All active database states in all execution paths of an active database program executing using the *Sequential* execution model are sequential database states.

Proof. The reader can verify using induction that such is the case. □

As a consequence of Corallary 1, we define a program correctness per its execution as defined by the *Sequential* execution model. An active database program is correct iff **all** eligible execution paths contain only sequential active database states and all possible quiescent states are producible by the *Sequential* execution model. Since Step 2 in *Sequential* is nondeterministic, the *Sequential* execution model may terminate in more than one quiescent state. We define $Correct_Y(\mathcal{D}_n,\mathcal{X})$ as the **set** of active database states reachable by all possible execution paths of $Sequential_Y(\mathcal{D}_n, \mathcal{X})$. We refer to the states in $Correct_Y(\mathcal{D}_n,\mathcal{X})$ as correct database states. More formally,

$$Correct_Y(\mathcal{D}_n,\mathcal{X}) = \{\text{states } S \mid \exists \text{ an execution of } Sequential_Y(\mathcal{D}_n, \mathcal{X}) \text{ that yields } S\} \quad \text{(EQ 5)}$$

Correct Active Database Program - Consider an active database program *Y* that begins in state $\mathcal{D}_i$ and processes the sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \ldots, \mathcal{X}_{n-1}\}$. Let the total set of possible quiescent states after sequentially processing all *n* events using the *Sequential* execution model be $Correct_Y(\mathcal{D}_i, \mathcal{X})$.

Definition: A program *Y* is *correct* under a execution model *X* iff

- All eligible execution paths contain only sequential database states, and
- if $X_Y(\mathcal{D}_n, \mathcal{X}) \Rightarrow \mathcal{D}_{n+k}$, then $\mathcal{D}_{n+k} \in Correct_Y(\mathcal{D}_n, \mathcal{X})$.

5. Serializability of Rules

We heavily exploit the standard definitions and methodology surrounding the serializability theorem [15]. However, its manifestation in active databases and our presentation requires some review and adaptation.

Serializability theory states the conditions upon which concurrent processing is equivalent to a serial interleaving of operations. A well known application of the theory is within database transaction models [15]. In [4], Bernstein states a set of conditions that specify when the order of interfering operations matter (RAW, WAW, and RAW). These conditions are violated when interfering operations are executed in parallel without restrictions. The results of violating the Bernstein conditions is that the database may move to an incorrect state.

Rule interference is synonymous to the Bernstien's conditions. In this case, the parallel execution of interfering rules may produce an incorrect state. Formally, for a MLM active database program, a rule R_0 *interferes* with a rule R_1 iff

$$\exists(Insert, T) \in A^{R_0} \mid \exists(Insert, T) \in Neg(C^{R_1}) \quad \text{(EQ 6)}$$

External events may also interfere with rules. An external event $\mathfrak{X}_0$ *interferes* with a rule R iff

$$\exists(Insert, T) \in \mathfrak{X}_0 \mid \exists(Insert, T) \in Neg(C^{R}) \quad \text{(EQ 7)}$$

In this paper, we use a simplified version of the bipartite dependency graphs developed in [16] to statically determine rule interferences[7]. A dependency graph G_m is defined as (V,E) where the vertices $V \in G_m$ represent either rules, represented as "circles" (○), or external events, represented as "diamonds" (◇). Edges $E \in G_m$, presented in Figure 2, represent the data dependency between rules and external events.

Edge	Validating Condition
1). R_0 —+→ R_1	R_1 monitors R_0; R_0 and R_1 do **not** interfere with one and another.
2). R_0 —-→ R_1	R_1 monitors R_0; R_0 and R_1 interfere with one and another.
3). $\mathfrak{X}_0$ —+→ R_0	R_0 monitors for events in $\mathfrak{X}_0$; $\mathfrak{X}_0$ and R_0 do **not** interfere with one and another.
4). $\mathfrak{X}_0$ —-→ R_0	R_0 monitors for events in $\mathfrak{X}_0$; $\mathfrak{X}_0$ and R_0 interfere with one and another.

FIGURE 2. Bipartite Graph Constructs

7. The simplifications result from the properties of MLMs.

Kuo et. al. identify a region in a dependency graph that may lead to erroneous behavior. They call this region a **cycle of interference** - a cycle in a dependency graph in which all edges are negative. The set of rules in a cycle of interference form a **mutual exclusion set**.

Kuo et. al. present two theorems based on rules within mutual exclusion sets. The theorems are based an the concepts of **cycle serializability** and **execution serializability** which are defined as follows: A parallel execution cycle c_k is cycle serializable iff there exists a serial execution of c_k, call this c_k*, such that execution of c_k in state ($\mathcal{D}_j$, $\mathcal{I}_j$) moves the database to a state ($\mathcal{D}_{j+1}$, $\mathcal{I}_{j+1}$) and the execution of c_k* in state ($\mathcal{D}_j$, $\mathcal{I}_j$) moves the database to a state ($\mathcal{D}^*_{j+m}$, $\mathcal{I}^*_{j+m}$) and $\mathcal{D}_{j+1} = \mathcal{D}^*_{j+m}$. An active database program with an execution path of n parallel execution cycles is execution serializable iff $\forall j \in [0, ..., n\text{-}1]$, cycle j is cycle serializable.

The theorems from [16] used in this paper are:

Cycle Serializability Theorem. The parallel execution of all the rules within a mutual exclusion set may not be *cycle serializable*. Proof is given in [16].

Serializability Theorem. A parallel execution cycle that does not contain all the rules within a mutual exclusion set is guaranteed to be *cycle serializable*. Proof is given in [16].

We use these theorems to establish concurrency schemes that force parallel execution cycles to become cycle serializable. As such, an active database execution path in which all execution cycles are cycle serializable contains only sequential database states.

6. Concurrency Schemes for MLM$^+$ Programs

This section presents the concurrency schemes for MLM$^+$ programs. We begin with a discussion on programs executing the *Parallel* execution model and conclude with the *Active* database execution model. Each section contains the three proofs of 1) the sufficient conditions for all execution cycles to be cycle serializable, 2) the sufficient conditions for confluence, and 3) the sufficient conditions for programs correctness. Our resulting concurrency schemes demonstrate that MLM$^+$ programs with all rules stated in E-C and C-A decoupled modes are correct and confluent.

6.1. Parallel Execution Model

6.1.1. Cycle Serializable

Lemma 1. Given a MLM$^+$ program in which all rules are specified in the E-C and C-A decoupled modes, all parallel execution cycles in all execution paths using the *Parallel* execution model are cycle serializable.

Proof. Instead of a direct proof of Lemma 1, it will suffice to prove the more general claim stating that for a MLM$^+$ program, all parallel execution cycles in all execution paths using the *Parallel* execution model are cycle serializable (regardless of coupling modes). Therefore, it will be vacuously true that the E-C decoupled and C-A decoupled modes are sufficient.

Step 0 in the parallel execution model is cycle serializable by definition. Now it is necessary to prove that Step 3 in the parallel execution model on a MLM$^+$ program is cycle serializable. Construct 1 in Figure 2 is the only edge notation connecting MLM$^+$ rules. Dependency graphs with

only positive edges contain no mutual exclusion sets. Kuo et. al.'s Serializability Theorem says that such *Parallel* execution cycles are guaranteed to be cycle serializable. Thus, a *Parallel* execution cycle containing any subset of rules within $\mathcal{R}$ is guaranteed to be cycle serializable and Step 3 must be cycle serializable. By induction, all execution cycles are cycle serializable and the claim has been proven. □

6.1.2. Confluence

Theorem 1. The execution of a MLM$^+$ program using the *Parallel* execution model in which all rules are specified in E-C and C-A decoupled modes is confluent.

Proof. Lemma 1 proves that all parallel execution cycles in all execution paths of an MLM$^+$ program executing the *Parallel* execution model (with the stated coupling modes) are cycle serializable. Therefore, MLM$^+$ programs executing the *Parallel* execution model are execution serializable by definition. Execution serializability implies that all execution paths are equivalent to some sequential execution path. As such, active database programs executing the *Parallel* execution model are equivalent to some sequential execution. [9] proves that sequential executions of MLM$^+$ programs are confluent, and therefore, MLM$^+$ programs using the *Parallel* execution model are confluent. □

6.1.3. Program Correctness

Theorem 2. The execution of a MLM$^+$ program using the *Parallel* execution model in which all rules are specified in E-C and C-A decoupled modes is correct.

Proof. Given any MLM$^+$ program *Y* in which all rules are stated in E-C and C-A decoupled modes, *Y* is correct under the *Parallel* execution model iff all parallel execution cycles in all execution paths are cycle serializable, and all executions of $Parallel_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for any initial state $\mathcal{D}_n$ and external event $\mathcal{X}$.

Lemma 1 satisfies the first conjunct by proving that all parallel execution cycles in all execution paths of *Y* (with the stated coupling modes) using the *Parallel* execution model are cycle serializable.

Now it is necessary to prove the second conjunct. [9] proves that sequential executions of MLM$^+$ programs are confluent. Lemma 1 proves that MLM$^+$ programs using the *Parallel* execution model are also confluent. Therefore, together the statements imply that $Parallel_Y(\mathcal{D}_n, \mathcal{X}) = Correcty_Y(\mathcal{D}_n, \mathcal{X})$.

Both conjuncts have been proven, and the theorem is satisfied. □

6.2. Active Database Execution Model

6.2.1. Cycle Serializable

Lemma 2. Given a MLM$^+$ program in which all rules are specified in the E-C and C-A decoupled modes, all parallel execution cycles in all execution paths using the *ActiveDatabase* execution model are cycle serializable.

Proof. Noting that external events add only constructs 3 and 5 to the dependency graphs, the proof of Lemma 2 is almost identical to that of Lemma 1. Specifically, it will suffice to prove the more general claim that all parallel execution cycles in all execution paths of a MLM$^+$ program using the *ActiveDatabase* execution model are cycle serializable (regardless of coupling modes).

Step 0 in the *ActiveDatabase* execution model is cycle serializable by definition. Now it is necessary to prove that Step 3 in the *ActiveDatabase* execution model on a MLM$^+$ program is cycle serializable. Constructs 1, 3 and 5 in Figure 2 are the only edges connecting MLM$^+$ rules executing the *ActiveDatabase* execution model. Dependency graphs with only positive edges contain no mutual exclusion sets. Kuo et. al.'s Serializability Theorem says that such parallel execution cycles are guaranteed to be cycle serializable. Thus, a parallel execution cycle containing any subset of rules within $\mathcal{R}$ and any number of external events is guaranteed to be cycle serializable and Step 3 must be cycle serializable. By induction, all execution cycles are cycle serializable and the claim has been proven. □

6.2.2. Confluence

Theorem 3. The execution of a MLM$^+$ program using the *ActiveDatabase* execution model in which all rules are specified in E-C and C-A decoupled modes is confluent.

Proof. Lemma 2 proves that all parallel execution cycles in all execution paths of an MLM$^+$ program executing the *ActiveDatabase* execution model (with the stated coupling modes) are cycle serializable. Therefore, MLM$^+$ programs executing the *ActiveDatabase* execution model are execution serializable by definition. Execution serializability implies that all execution paths are equivalent to some sequential execution path. As such, active database programs executing the *ActiveDatabase* execution model are equivalent to some sequential execution. [9] proves that sequential executions of MLM$^+$ programs are confluent, and therefore, MLM$^+$ programs using the *ActiveDatabase* execution model are confluent. □

6.2.3. Program Correctness

Theorem 4. The execution of a MLM$^+$ program using the *ActiveDatabase* execution model in which all rules are specified in E-C and C-A decoupled modes is correct.

Proof. Given any MLM$^+$ program Y in which all rules are stated in E-C and C-A decoupled modes, Y is correct under the *ActiveDatabase* execution model iff all parallel execution cycles in all execution paths are cycle serializable, and all executions of $ActiveDatase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for any initial state $\mathcal{D}_n$ and sequence of external events $\mathcal{X}$.

Lemma 3 satisfies the first conjunct by proving that all parallel execution cycles in all execution paths of *Y* (with the stated coupling modes) using the *ActiveDatabase* execution model are cycle serializable.

Now it is necessary to prove the second conjunct. [9] proves that sequential executions of MLM^+ programs are confluent. Lemma 1 proves that MLM^+ programs executing the *ActiveDatabase* execution model are also confluent. Therefore, together the statements imply that $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) = Correcty_Y(\mathcal{D}_n, \mathcal{X})$.

Both conjuncts have been proven, and the theorem is satisfied. □

7. Concurrency Schemes of MLM^- Programs

We now consider MLM^- programs. In general, MLM^- programs are not confluent since rules and external events may interfere with one and another [23,28].

Our definition of program correctness stipulates that a program is correct iff the quiescent state is reproducible by the *Sequential* execution model. With regard to ordering, the *Sequential* execution model sequentially processes each external event until quiescence. Consequently, in addition to presenting the sufficient conditions for cycle serializability, we present the sufficient conditions to maintain the ordering of external events[8].

We begin with a discussion on event sequencing.

7.1. Event Sequencing

To define event sequencing, we must present two new definitions. First, we say that a set of rules, *A*, executes in **isolation** from another set of rules *B*, when there does not exist a rule within either *A* or *B* that interferes with one and another. Secondly, for a set of rules $R \subseteq \mathcal{R}$, we define the set of rules within the **Closure(R)** as follows:

rule R_0
E: { (*Insert*,T_0) }
C: $\neg T_0(3)$
A: { (*Insert*,$T_1(2)$) }

$\mathcal{X}_0$ = { (*Insert*,$T_0(1)$) }

$\mathcal{X}_1$ = { (*Insert*,$T_0(3)$) }

$\mathcal{D}_0$ = {}

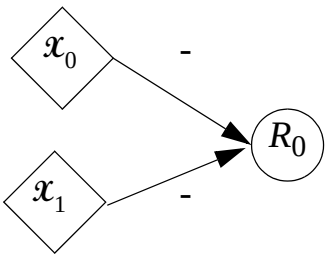

FIGURE 3. External Event Sequencing.

8. This sequencing has not been necessary in our previous sections. In Section 6, we presented MLM^+ programs that are confluent. Confluence means sequence is irrelevant.

Algorithm: *Closure*
Input: $R \subseteq \mathcal{R}$
Ouput: $S \subseteq \mathcal{R}$

$S \leftarrow \{R\}$
Repeat until *S* is unchanged:

$$S \leftarrow S \cup X \in \mathcal{R} \mid X \in Triggers(A^{Y}) \text{ for some } Y \in S.$$

Graphically, the *Closure*(*R*) contains all rules reachable by a depth first traversal of the dependency graph starting from the rules in set *R*.

We now define **event sequencing**. Event $\mathcal{X}_0$ is processed in *sequence* before an event $\mathcal{X}_1$ iff all rules triggered by the $Closure(Triggers(\mathcal{X}_0))$ are evaluated before (in time) or in *isolation* from the rules triggered by $Closure(Triggers(\mathcal{X}_1))$.

The *closure* is interesting since the *Sequential* execution model processes an external event $\mathcal{X}$ by locking the active database and evaluating all the rules triggered in the $Closure(Triggers(\mathcal{X}))$ until quiescence. Therefore, a proof of program correctness for an execution model must prove that all external events are sequenced.

The following example demonstrates the necessity for external event sequencing.

Example 1. Consider Figure 3. Let $\mathcal{X}_0$ be applied in $\mathcal{D}_0$ and $\mathcal{X}_1$ be applied in $\mathcal{D}_2$. As such, $\mathcal{X}_0$ triggers R_0 in $\mathcal{D}_0$ and $\mathcal{X}_1$ triggers R_0 again in $\mathcal{D}_2$. If R_0 is stated in E-C and C-A immediate modes, R_0 will be evaluated in $\mathcal{D}_1$. $C^{R_0}(\mathcal{D}_1) \equiv true$ and R_0 will insert the number 2 to T_1. Upon $\mathcal{X}_1$, R_0 will again be evaluated in $\mathcal{D}_3$. $C^{R}(\mathcal{D}_3) \equiv false$ and R_0 will not fire a second time. The resulting quiescent state will be $\mathcal{D}_3 = \{T_0(1), T_0(2), T_1(3)\}$.

Now, consider R_0 in E-C and C-A decoupled mode. In this case, $\mathcal{X}_0$ will trigger R_0 in $\mathcal{D}_0$ and $\mathcal{X}_1$ may trigger R_0 again in $\mathcal{D}_1$. Due to R_0's E-C decoupled mode, the database scheduler may delay condition evaluation of R_0 until after $\mathcal{D}_1$. Therefore, $\mathcal{X}_1$ will be applied in $\mathcal{D}_2$ and $\mathcal{D}_2 = \{T_0(1), T_0(3)\}$, $C^{R_0}(\mathcal{D}_2) \equiv false$ and R_0 will not execute its action. The final state after processing will be $\mathcal{D}_x = \{T_0(1), T_0(3)\}$, $x > 2$. This execution is not consistent with any sequential application of the external events, and therefore, is incorrect.

7.2. Parallel Execution Model

7.2.1. Cycle Serializability

Lemma 3. Given an MLM$^-$ application, all parallel execution cycles in all execution paths using the *Parallel* execution model are cycle serializable under the following condition:

- At least one rule within all mutual exclusion sets is specified in E-C and C-A immediate modes.

rule R_0
E: { (*Insert*,T_0) }
C: $\neg T_0(1)$
A: { (*Insert*,$T_1(1)$) }

rule R_1
E: { (*Insert*,T_1) }
C: $\neg T_1(1)$
A: { (*Insert*,$T_0(1)$) }

$\mathcal{D}_0$ = { }
$\mathcal{X}_0$ = { (*Insert*,$T_0(3)$),
(*Insert*,$T_1(3)$) }

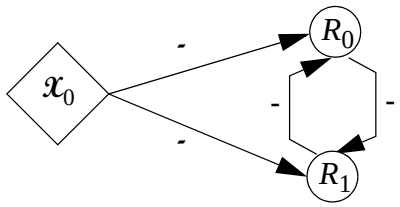

FIGURE 4. An example cycle of dependency.

Proof. The proof of Lemma 3 uses the same logic as the proof of Lemma 1.

Execution begins in Step 0 of the *Parallel* execution model. Step 0 is cycle serializable by definition. Now it is necessary to prove that Step 3 in the *Parallel* execution model on a MLM$^-$ program is cycle serializable.

Consider a parallel execution cycle that contains all the rules within a mutual exclusion set. Lemma 3's condition sets at least one these rules to E-C and C-A immediate modes. According to the definitions in Section 4.1, rules become atomic when they are set to E-C and C-A immediate modes. Atomic operations take the necessary locks to execute serially in the face of conflicting operations. Therefore, the mutual exclusion set will not truly execute in parallel if interference occurs. Kuo et. al.'s Serializability Theorem tells us that such parallel execution cycles are cycle serializable. Thus, a parallel execution cycle containing any subset of rules within $\mathcal{R}$ is guaranteed to be cycle serializable and Step 3 must be cycle serializable. By induction, all execution cycles are cycle serializable, and the claim has been proven. □

Lemma 3 gives rise to the following corollary.

Corollary 2. Given an MLM$^-$ application, all parallel execution cycles in all execution paths using the *Parallel* execution model are guaranteed to be cycle serializable, by static methods, only under the following condition:

- At least one rule within all mutual exclusion sets is specified in E-C and C-A immediate modes.

Proof. For the sake of contradiction, suppose that a weaker concurrency scheme exists that guarantees cycle serializability within the *Parallel* execution model. The next two concurrency schemes that are weaker than the E-C and C-A immediate modes are the E-C immediate and C-A deferred modes and the E-C deferred and C-A immediate modes.

First, consider the mutual exclusion set consisting of n rules in which one rule is stated in E-C immediate and C-A deferred modes with the remaining rules stated in E-C and C-A decoupled modes. In the current case, a loose maximum bound for the possible number of legal interleaving operations in which the E-C immediate rule's condition must be evaluated first is $O(2n\text{-}1)!$. A parallel execution cycle may choose one such interleaving that is equivalent to evaluating the E-C immediate rule's condition first, followed by the remaining rules' conditions in any order, followed by the rules' actions in any order. This interleaving is equivalent to executing all rules in parallel since all rules will evaluate their respective conditions in the same database state and execute their actions accordingly. Kuo et. al.'s Cycle Serializability Theorem says that the parallel

execution of all rules in a mutual exclusion set may not be serializable. This is a contradiction to the claim.

Now consider the mutual exclusion set consisting of *n* rules in which one rule is stated in E-C deferred and C-A immediate modes with the remaining rules stated in E-C and C-A decoupled modes. In this case, a loose maximum bound for the possible number of legal interleaving operations in which the C-A immediate rule's action must be executed before any other rule action is $O(2n\text{-}1)!$. A parallel execution cycle may choose one such interleaving that is equivalent to evaluating all E-C decoupled rules' conditions first, followed by the E-C deferred rule's condition, followed by the C-A immediate rule's action, followed by the remaining rules' actions in any order. This interleaving is equivalent to executing all rules in parallel since all rules will evaluate their respective conditions in the same database state and execute their actions accordingly. Kuo et. al.'s Cycle Serializability Theorem says that the parallel execution of all rules in a mutual exclusion set may not be serializable. Consequently, this also is a contradiction to the claim.

Therefore the only concurrency model that guarantees cycle serializability by static methods is the one stated in the corollary's condition. □

Example 2. To illustrate Corallary 2, consider the example presented in Figure 4. In the figure, $R_0, R_1 \in \mathcal{R}$, $T_0, T_1 \in \mathcal{E}$ and R_0 and R_1 form a cycle of interference. Consider when R_0 is stated in E-C and C-A decoupled modes and R_1 is stated in E-C immediate and C-A deferred modes. Let the external event $\mathcal{X}_0$ insert the tuple 3 into both T_0 and T_1. Due to coupling mode semantics, R_1's condition will immediately be evaluated while R_1's action will be evaluated right before transaction commit. In the *Parallel* execution model, no other external events occur during R_0 and R_1's conditions and actions executions. Therefore, a legal interleaving of operations is to apply $\mathcal{X}_0$ in state $\mathcal{D}_0$, evaluate C^{R_0} in $\mathcal{D}_1$, evaluate C^{R_1} in $\mathcal{D}_2$, and execute A^{R_0} and A^{R_1} in state $\mathcal{D}_3$. Thus, both $C^{R_0}(\mathcal{D}_1)$ and $C^{R_1}(\mathcal{D}_2)$ evaluate to *true*. Upon the completion of R_0 and R_1's actions, the database will be in an inconsistent state; specifically, both R_0 and R_1 executed in parallel inserting the tuple 1 into both T_0 and T_1.

7.2.2. Program Correctness

Theorem 5. The execution of an MLM$^-$ application using the *Parallel* execution model obeying the following condition is correct:

- At least one rule in every mutual exclusion set uses the E-C and C-A immediate modes.

Proof. The proof of Theorem 5 is similar in logic the proof of Theorem 4. Specifically, given any MLM$^-$ program *Y* in which all rules are stated in E-C and C-A decoupled modes, *Y* is correct under the *Parallel* execution model iff all parallel execution cycles in all execution paths are cycle serializable, and all executions of $Parallel_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for any initial state $\mathcal{D}_n$ and external event $\mathcal{X}$.

Lemma 3 satisfies the first conjunct by proving that all parallel execution cycles in the execution of *Y* (with the stated coupling modes) using the *Parallel* execution model are cycle serializable.

Now it is necessary to prove the second conjunct. By definition, the *Parallel* execution model processes each event until quiescence. Lemma 3 tells us that all parallel execution cycles are cycle

serializable. Thus, all events are processed sequentially and for any two external events $\mathcal{X}_0$ occurring before $\mathcal{X}_1$, the $Closure(Triggers(\mathcal{X}_0))$ is evaluated to quiescence before the $Closure(Triggers(\mathcal{X}_1))$ by definition. We can therefore conclude that for all executions of *Y*, $Parallel_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$. [9]

Both conjuncts have been proven, and the theorem is satisfied. □

7.3. ActiveDatabase Execution Model

7.3.1. Cycle Serializability

Lemma 4. Given an MLM- application, all parallel execution cycles in all execution paths using the *ActiveDatabase* execution model are cycle serializable under the following condition:

- At least one rule within all mutual exclusion sets is specified in E-C and C-A immediate modes.

Proof. The proof of Lemma 4 is similar to the proof of Lemma 3. Specifically, the only difference between MLM- programs using the *ActiveDatabase* execution model versus the *Parallel* execution model, with respect to cycle serializability, is that parallel execution cycles may contain external events. Yet, external events do not introduce nonserializable behavior. This is because external events are atomic. Atomic operations take the necessary locks to execute serially in the face of conflicting operations. Therefore, a mutual exclusion set containing an external event cannot truly be executed in parallel if interference occurs[10]. Thus, by Kuo et. al.'s Serializability Theorem and the same reasoning as Lemma 3, all parallel execution cycles using the *ActiveDatabase* execution model under the stated condition are cycle serializable. □

7.3.2. Program Correctness

We present three sets of sufficient conditions for the correctness of programs executing the *ActiveDatabase* execution model. Each successive set allows for more concurrency.

The first and *unnecessarily* restrictive concurrency model is presented in Theorem 6.

Theorem 6. The execution of a MLM- program using the *ActiveDatabase* execution model obeying the following conditions is correct.

- At least one rule in every mutual exclusion set uses the E-C immediate and C-A immediate modes.

9. In fact, the sufficient conditions for confluent MLM- programs are shown in [9,19]. These studies demonstrate the transformations upon *stratified* active database rules (programs that contain no cycles of interference) to obtain confluence.

10. Further, external events are never part of a cycle of dependency. The in-degree of all external event nodes in a bipartite graph is 0.

- For every external event $\mathcal{X}$ in which the *Closure*($\mathcal{X}$) contains a rule connected with a negative edge in the dependency graph, all the rules in the *Closure*($\mathcal{X}$) are stated in E-C and C-A immediate modes.

Proof. The proof of Theorem 6 is similar in logic the proof of Theorem 5. Specifically, given any MLM$^-$ program *Y* in which all rules are stated in the coupling modes as dictated by the theorem conditions, *Y* is correct under the *ActiveDatabase* execution model iff all parallel execution cycles in all execution paths are cycle serializable, and all executions of $ActiveDatase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for any initial state $\mathcal{D}_n$ and any sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \mathcal{X}_1, \ldots \mathcal{X}_{n-1}\}$.

Lemma 4 satisfies the first conjunct by proving that all parallel execution cycles in the execution of *Y* (with the stated coupling modes) using the *ActiveDatabase* execution model are cycle serializable.

Now it is necessary to prove the second conjunct. The theorem conditions specify that for every external events $\mathcal{X}$ in which the *Closure*($\mathcal{X}$) contains a rule connected with a negative edge in the dependency graph, all the rules in the *Closure*($\mathcal{X}$) are stated in E-C and C-A immediate modes. Therefore, such events are necessarily sequenced since all the rules in which $\mathcal{X}$ may trigger are executed atomically.

Now consider the remaining external events. These events do not trigger any interfering rules. By Theorem 3, these subregions of the dependency graph are confluent (they are MLM$^+$ regions) and are necessarily executed in isolation. Thus, under the theorem conditions $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for all possible execution paths.

Both conjuncts have been proven, and the theorem is satisfied. □

Though sufficient, Theorem 6 is a very restrictive concurrency model. For one, Theorem 6 does not take into account transaction boundaries. Our definition of external events are that they are atomic and committed. Theorem 7 exploits this property and weakens the concurrency model accordingly.

Theorem 7. The execution of a MLM$^-$ program using the *ActiveDatabase* execution model obeying the following conditions is correct.

- At least one rule in every mutual exclusion set uses the E-C immediate and C-A immediate modes.
- For every external event $\mathcal{X}$ in which the *Closure*($\mathcal{X}$) contains a rule connected with a negative edge in the dependency graph, all the rules in the *Closure*($\mathcal{X}$) are stated in E-C and C-A deferred modes or stronger.

Proof. This proof is similar in logic to the proof of Theorem 6. Specifically, Lemma 4 says that the first condition is sufficient for cycle serializability. Therefore, it is only left to prove that the loosened conditions of Theorem 7 are sufficient to sequence external events.

The theorem conditions specify that for every external event $\mathcal{X}$ in which the *Closure*($\mathcal{X}$) contains a rule connected with a negative edge in the dependency graph, all the rules in the *Closure*($\mathcal{X}$) are stated in E-C and C-A deferred modes or stronger. Therefore, the events triggering rules within the *Closure*($\mathcal{X}$) that contain a negative edge are necessarily sequenced since all the rules are executed to quiescence before the end of the transaction, and no other external event executes until the transaction has been completely committed (by the definition of an external event).

Now consider the remaining external events. These events do not trigger any interfering rules. By Theorem 3, these subregions of the dependency graph are confluent (they are MLM$^+$ regions) and are necessarily executed in isolation. Thus, under the theorem conditions $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for all possible execution paths.

Both conjuncts have been proven, and the theorem is satisfied. □

Theorem 7 provides more concurrency than Theorem 6 since external events that trigger interfering rules do not have to execute all rules within the closure atomically. Therefore, rule execution from other regions in the dependency graph may continue processing in parallel. Yet the conditions in Theorem 7 can still be weakened. A close examination of rule dependency graphs reveals that external events need to be sequenced only when rules within their closures interfere with one and another.

Towards this end, we define **external event interference**. We say that two external events, $\mathcal{X}_i$ and $\mathcal{X}_j$, interfere with one and another when $\{Closure(\mathcal{X}_i) \cap Closure(\mathcal{X}_j)\} = A$, $A \neq \varnothing$, and $\exists R \in A$ such that R is a rule connected, in either direction, with a negative edge in the dependency graph. Lemma 5 follows from this definition.

Lemma 5. External events that do not interfere with one and another may be executed in parallel without violating event sequencing.

Proof. The proof is by contradiction. Consider the case when two external events $\mathcal{X}_i$ and $\mathcal{X}_j$, where $i < j$, must be sequenced but do not interfere with one and another. External events need to be sequenced when a rule that is triggered from within a later external event's closure invalidates a rule from an earlier external event's closure before the earlier external event evaluates to quiescence. In other words, $\mathcal{X}_i$ and $\mathcal{X}_j$ are not sequenced when some rule $Z \in Closure(\mathcal{X}_i)$ contains a negated variable that may be invalidated by the rules triggered in the *Closure*($\mathcal{X}_j$) before Z is evaluated by the rules triggered in the *Closure*($\mathcal{X}_i$).

In MLM programs, Z is invalidated by the rules within *Closure*($\mathcal{X}_j$) iff Z contains a rule connected with a negative edge in the dependency graph and $Z \in Closure(\mathcal{X}_j)$.[11] Thus, $Z \in \{Closure(\mathcal{X}_i) \cap Closure(\mathcal{X}_j)\}$ and $\mathcal{X}_i$ and $\mathcal{X}_j$ interfere. This is a contradiction proving that external events that do not interfere with one and another may be executed in parallel without violating event sequencing. □

We are now ready to present our loosest concurrency model for MLM$^-$ program using the *ActiveDatabase* execution model.

Theorem 8. The execution of a MLM$^-$ program using the *ActiveDatabase* execution model obeying the following conditions is correct:

- At least one rule in every mutual exclusion set uses the E-C immediate and C-A immediate modes.

11. We assume that all applicable rules evaluate on all events (Section 2.2.1). This assumption implies that $Z \in$ *Closure*($\mathcal{X}_j$).

- For every external events $\mathcal{X}$ that *interferes* with another external event, all the rules in the *Closure*($\mathcal{X}$) are stated in E-C and C-A deferred modes or stronger.

Proof. This proof is similar to the proof of Theorem 7. Specifically, Lemma 4 says that the first condition is sufficient for cycle serializability. Therefore, it is only left to prove that the loosened conditions of Theorem 8 are sufficient to sequence external events.

The proof of the sequencing of external events is by contradiction. Consider the case when two external events $\mathcal{X}_i$ and $\mathcal{X}_j$, where $i > j$, obey the conditions of Theorem 8 are not sequenced. Lemma 5 says that external events that do not interfere with one and another are necessarily sequenced. Thus, $\mathcal{X}_i$ and $\mathcal{X}_j$ must interfere with one and another to violate event sequencing.

The theorem conditions specify that all the rules in the $Closure(\mathcal{X}_i)$ are state in E-C and C-A deferred modes or stronger when $\mathcal{X}_i$ and $\mathcal{X}_j$ interfere with one and another. Therefore, all the rules triggered in the *Closure*($\mathcal{X}_i$) are evaluated to quiescence before the rules triggered in the *Closure*($\mathcal{X}_j$) since $\mathcal{X}_j$ must occur in the next transaction by definition. This forms a contradiction that says $\mathcal{X}_i$ and $\mathcal{X}_j$ must be sequenced. Thus, under the theorem conditions, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for all possible execution paths.

Both conjuncts have been proven, and the theorem is satisfied. □

8. Conclusion and Future Work

The large number of coupling modes developed to integrate active rules within the ACID properties of a database do not easily scale. Coupling mode semantics become unmanageable within applications that are classified as hard rule systems - rule programs that have a large number of interacting rules that search through lots of data. This paper presents the resulting simplifications of a significant subclass of hard rule systems we call Monotonic Log Monitor programs, active database programs in which tuples are inserted, but never updated nor deleted. Our results demonstrate that the number of applicable coupling modes are significantly reduced for programs obeying the MLM restrictions which in turn minimizes the necessary coupling mode support of the underlying database system.

A separate contribution of our work is that our constructive proof techniques may be exploited to build a compiler-based concurrency implementation system. The compiler would operate by constructing a bipartite rule dependency graph from an input rule program. The compiler then would walk the graph and output the concurrency schemes presented in this paper.

It is our belief that the details of transaction models and concurrency schemes are application dependent. This paper represents the first step in such a general purpose system. As the number of investigated problem areas are expanded, a compiler-based systems may be presented, computationally or otherwise, with a problem type and implement the appropriate isolation model. We believe that such technology attacks one of the major complexity stumbling blocks to general use of active database systems by insulating application programmers from the details of database integration.

Our constructive proof techniques utilize the results in confluent rule system, dependency graph, and serializability theories. These techniques are readily adapted for formal analysis. However, we believe further parallelism may be gained through optimizing rule-based rewrites and dynamic analysis such as the ones presented in [16]. Dynamic analysis allows for coupling mode assignments based on the semantic relationships that are only available at runtime (implying dynamic

assignment of coupling modes). The net result may allow for closures of rules to be stated in a weaker coupling modes than could have otherwise have been discerned. We leave this area of work open.

9. Acknowledgments

I would like to thank Lance Obermeyer, Vasilis Samoladis, and Francois Barboncon for their assistance. Their insightful suggestions and discussion tremendously added to this work.

10. Bibliography

1 A. Aiken, J. Widom, J.M. Hellerstein, “Behavior of Database Production Rules: Termination, Confluence, and Observable Determinism”, *Proceedings of the ACM-SIGMOD Conference*, 1992.

2 E. Baralis, S. Ceri, and S. Paraboschi, “Modularization Techniques for Active Rules Design,” *ACM Transaction on Database Systems*, vol. 21. No. 1. March 1996.

3 E. Baralis and J. Widom, “Using Delta Relations to Optimize Condition Evaluation in Active Databases,” *Rules in Database Systems*, 1995.

4 P.A. Bernstein and N. Goodman, “Concurrency Control in Distributed Database Systems,” *ACM Computing Surveys*, vol 13, no. 2, June 1981.

5 J.C. Browne et al, “Modularity in rule-based programming,” *International Journal on Artificial Intelligence Tools*, vol. 4, no. 1&2, pp. 201-218, 1995.

6 A. Buchmann, J. Zimmermann, J. Blakeley, “Building an Integrated Active OODBMS: Requirements, Architecture, and Design Decisions”, *Proceedings of the 11th International Conference on Data Engineering.* Taipeh, Taiwan, March, 1995.

7 C. Bussler and S. Jablonski, “Implementing agent coordination for workflow management systems using active database systems,” *Proceedings of the 4th International Workshop on Research Issues in Data Engineering*. Houston, Texas, February, 1994.

8 S. Chakravarthy, “Snoop: An expressive event specification language for active databases,” *Knowledge and Data Engineering Journal*, vol. 14, pp. 1-26, 1994.

9 S. Comai, L. Tanca, “Using the Properties of Datalog to prove Termination and Confluence in Active Databases,” *Rules in Database Systems*, Skövd, Sweden, June 1997.

10 S. Correl and D. Miranker, “On isolation, concurrency, and the Venus rule language,” in *Proceedings of the 4th International Conference on Information and Knowledge Management*. Baltimore, MD, November, 1995

11 J. Crawford, D. Dvorak, D. Litman, and A. Mishra, “Device-structured diagnosis: A middle ground,” unpublished manuscript, 1994.

12 U. Dayal, A. P. Buchmann, and S. Chakravarthy, “The HiPAC Project,” in *Active database systems: triggers and rules for advanced database processing*, J. Widom and S. Ceri, Eds. San Francisco, CA.: Morgan Kaufmann Publishers, 1996, pp. 177-205.

13 L. Do and P. Drew, “Active Database Management of global Data Integrity Constraints in Heterogeneous Database Environments,” *Proceedings of the 11th International Conference on Data Engineering*, Taipei, Taiwan, March, 1995.

14 T. Ishida and S.J. Stolfo, “Simultaneous firing of production rules on tree structured machines,” *International Conference on Parallel Processing*, 1998.

15 H. Korth, A. Silberschatz. *Database System Concepts*. McGraw-Hill, Inc. 1991.

16 C.M. Kuo, D.P. Miranker, and J. C. Browne, “On the Performance of the CREL System,” *Journal of Parallel and Distributed Computing*, vol 13, 1991.

17 L. Obermeyer, “Abstractions and Algorithms for Active Multidatabases,” Doctoral Thesis, University of Texas at Austin, 1999.

18 P. Picouet and V. Vianu, “Semantics and expressiveness issues in active database,” in *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, San Jose, CA, May 1995.

19 L. Raschid, “Maintaining Consistency in a Stratified Production System Program,” *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI)*, pp. 284-289, 1990.

20 S. Stolfo et. al, *The ALEXSYS mortgage pool allocation expert system: A case study of speeding up rule-based programs*. Columbia University Department of Computer Science and Center for Advanced Technology, 1990.

21 M. Stonebraker, “The integration of rule systems and database systems,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 4, no. 5, October, pp. 415-423, 1992.

22 M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos, “On rules, procedures, caching and views in database systems,” *Proceedings of the 1990 ACM SIGMOD Conference on Management of Data*. Atlantic City, N.J., May, 1990.

23 J. D. Ullman. *Principles of Database and Knowledge-Based Systems*. W H Freeman & Co, 1988.

24 L. Warshaw, et. al., “Monitoring Network Logs for Anomalous Activity,” Applied Research Laboratories at the University of Texas at Austin, technical report #TP-99-1, 1998.

25 L. Warshaw, D.P. Miranker, “A case study of Venus and a declarative basis for rule modules,” *Proceedings of the 5th Conference on Information and Knowledge Management*, November, 1996.

26 L. Warshaw, L. Obermeyer, D. P. Miranker, “VenusIDS: An Active Database Component for an Intrusion Detection System,” Applied Research Laboratories at the University of Texas at Austin, technical report # , 1999.

27 J. Widom, “The Starburst Active Database Rule System,” *IEEE Transactions on Knowledge and Data Engineering*, August, 1996.

28 J. Widom and S. Ceri, *Active Database Systems: Triggers and Rules for Advanced Database Processing*, Morgan Kaufmann Publishers, 1996.