Facilitating Hard Active Database Applications

by

Lane Bradley Warshaw, B.S., M.S.

Dissertation

Presented to the Faculty of the Graduate School of

the University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2001

Facilitating Hard Active Database Applications

Approved by
Dissertation Committee:

_____
Daniel P. Miranker, Supervisor

_____

_____

_____

_____

_____

Dedication

To my loving family.

Acknowledgements

Completing a dissertation requires patience and careful attention from many people. However, two people deserve special mention. First, Professor Daniel P. Miranker provided continual support and encouragement even while starting a new business. Second, Dr. Lance Obermeyer provided inspiration and a helpful ear with many portions of this research. Without his contributions, this dissertation would not have been possible. I also would like to recognize the Applied Research Laboratory where much of the dissertation's application work was performed. In particular, I would like to thank Sara Matzner for her support and John Williams for his hands-on help with the implementation of the VenusDB optimizer. I would also like to thank Francois Barbancon and Vasilis Samoladis for their careful reviews of Chapter 4. Tom Mathieu and Lisa Mathieu also deserve special recognition for their assistance in editing the dissertation document.

Facilitating Hard Active Database Applications

Publication No._____

Lane Bradley Warshaw, Ph.D.
The University of Texas at Austin, 2001

Supervisor: Daniel P. Miranker

Active database technology enhances traditional databases with rules that are executed in response to database events. This enhancement promises exceptional returns. Useful applications of the technology include view maintenance, workflow, and real-time decision control systems.

Unfortunately, penetration of active databases has largely been restricted to simple rule systems. This narrowed focus can be attributed to an explicit connection of the active rules to the underlying concurrency control system. Although this explicit connection provides a level of flexibility, it becomes semantically intractable in more complex applications. Furthermore, rule execution is computationally intensive. Since rules spawn the execution of many queries over the contents of the database, substantial skill is required to develop complex applications without introducing long duration transactions.

This dissertation addresses three issues surrounding application development that inhibits the feasibility of active databases. First, a quantitative evaluation is performed on the semantics of VenusDB, a modular active database language that executes within the nested transaction model. This evaluation provides evidence that rule modules improve system maintainability. Second, the most general contribution of this dissertation is the identification and study of Log

Monitoring Applications (LMAs). LMAs are expert system applications that analyze logs maintained in a database. This dissertation develops the formal execution semantics and correctness proofs of concurrency schemes for applications within the LMA class. The results show that only a minimal number of coupling modes are necessary for the database integration of hard rule systems obeying the LMA restrictions. Third, the architecture and evaluation of an active database optimizer is presented. This optimizer uses database statistics to optimize rules as well as suggest physical schema optimizations. The optimizer is integrated within VenusDB to improve system scalability.

Table of Contents

Chapter 1 Introduction

Active databases are databases augmented with production rules that are evaluated only on specified events [22,36,94]. Such production rules, coined Event-Condition-Action rules (ECA rules), have been the subject of investigation since the early 1980's. This early work gained impetus from the heavy exposure of rule programming associated with Japan's Fifth Generation Project. At that time, AI research was developing expert systems that exploited databases for input. It quickly became apparent that many of the core database services could be more effectively implemented using active rules [98]. As a result, active database research focused on applications such as integrity constraints and view maintenance. As the work matured, new applications, such as workflow systems, appeared. Workflow systems, programs that automate the movement of information in the business process, present interesting transaction and persistent challenges [15].

AI research, on the other hand, was evolving to solve complex search related problems. This research concentrated on improving the performance of rule matching algorithms [17,39,63] and parallel computation [50,60,63]. Active database applications that necessitated rule processing of this type leveraged this work by replicating its data within an expert system shell[1]. These loosely coupled systems suffer from the inefficiency of copying and reformatting data, wasted space, and data consistency problems.

Citing the problems of loosely coupled systems, recent work has focused on implementing complex rule applications using active database technology

1. Haley Enterprise's Rete++ and GenSym's G2 are two examples of commercial expert systems that provide hooks for copying data from a database.

[69,86]. This dissertation addresses issues that inhibit the development of such complex applications.

The complexity of applications is explained in a two-axis active database application taxonomy segregated by the complexity of the program's rule systems. On one axis of this taxonomy, Micheal Stonebraker proposed a classification based on the amount of **search** by the applications' rule systems [86]. *Simple rule systems* have few rules with little interaction, while *hard rule systems* have many rules with significant interaction. On the other axis, Lance Obermeyer suggested classifying problems on the amount of the **data** being searched [69]. This dimensional taxonomy yields four distinct regions (Figure 1).

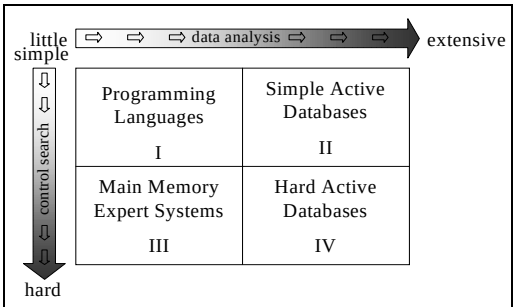


FIGURE 1. Two Axis Taxonomy of Active Databases

Region I consists of simple rule systems that investigate small amounts of data. Applications within this region are implemented using standard programming languages.

Region II consists of simple rule systems that investigate large amounts of data. It is within this region that early active database research was focused [33, 97]. Region II applications require the sophisticated data retrieval methods of standard databases, but do not necessitate robust rule engines for complex searching. Representative applications include core system services such as view maintenance, integrity constraints, and workflow systems [3,15,22,36,94].

Region III consists of hard rule systems that investigate small amounts of data. The minimal data requirement of this region alleviates the need for the database facilities of decision support and concurrency control. As such, region III

applications are implemented using expert system languages such as the OPS family of rule languages, CLIPS, and Venus [18,40,44].

Lastly, region IV applications consist of hard active database applications. These applications consists of hard rule systems that investigate an extensive amount of data. The complexity of these problems requires robustness from both the rule inference engine and the data retrieval utility. Representative applications include network security monitors and real-time decision control systems [85,94,95] (See Appendix).

The goal of this research is to facilitate hard active database applications by abstracting language limitations that inhibits their development. The contributions of this dissertation are threefold.

First, the previous generations of active database languages were designed for Region II applications in which individual rules are executed as independent programs. These languages contain operationally defined semantics with no primitives for rule organization [37,78,100]. As such, these early languages do not scale to hard active database development. This dissertation addresses language semantics by detailing a quantitative evaluation of the semantics of the active database language VenusDB as it relates to code complexity. VenusDB is an active database language that extends the traditional expert system model by including facilities for operating directly on database tables as well as a formal definition for rule modules that execute within the nested transaction model. This evaluation provides evidence that these language facilities improve system maintainability.

Second, ECA rules are not simply production rules applied to data within a database; rule computation must be integrated within the decision and concurrency control systems of the underlying database. The accepted method of rule integration, introduced by the HiPAC project, is for active database developers to explic-

itly specify transaction behavior via a pair of *coupling modes* [33]. The modes specify the transaction relationship of 1) database events to condition evaluation and 2) condition evaluation to action execution. This method has been effective in integrating Region II applications [86]. However, ongoing research has lead to the development of dozens of coupling modes [21,24,33,98]. As a result, coupling modes often burden application programmers with extremely difficult conceptual specifications, and they have proven to be one of the most difficult conceptual obstacles in the development of hard active database applications.

It is the author's belief that the details of transaction models and concurrency schemes are application dependent. In support of this hypothesis, the most general contribution of this dissertation is to begin insulating application programmers from the complexities introduced by coupling modes. In the course of developing the applications for this dissertation, a class of applications called Log Monitoring Applications (LMAs) became evident. LMAs are a subclass of hard rule systems that analyze logs maintained in a database. This dissertation defines formal execution semantics for the LMA class and presents correctness proofs of concurrency schemes for LMAs. The results demonstrate that only a minimal number of coupling modes are necessary for the database integration of hard rule systems obeying the LMA restrictions. Further, since the correctness proofs are constructive, the correctness theorems themselves may be used to form a compiler-based rewrite system that specifies concurrency schemes. This dissertation represents the first step in such a general purpose system that would completely isolate application programers from the details of concurrency control.

Third, active database applications are computationally intensive. In essence, each rule executes one or more queries over the contents of the database. In the context of hard active database applications, it is essential to optimize the database design for scalable performance. Current state-of-the-art active databases

fall short of this goal by relying on the underlying database optimizers that only optimize individually executed queries [74,98]. Consequently, physical schema optimizations are left to the database administrators. Within hard active databases, this decision requires administrators to analyze database workloads, the structure of all rules, as well as the columns of all tables in an unfamiliar domain. This dissertation addresses these issues by introducing the architecture and evaluation of the VenusDB optimizer. The VenusDB optimizer uses databases statistics to couple the queries to be executed on component databases with suggestions for physical schema optimizations. Together, these techniques assist database administrators in deploying scalable systems.

This dissertation is organized as follows. Chapter 2 presents background information. Chapters 3-5 present the main contributions of the dissertation. Specifically, Chapter 3 introduces VenusDB and details the language evaluation; Chapter 4 introduces Log Monitoring Applications and presents the concurrency schemes for such applications; and Chapter 5 describes the VenusDB optimizer and its empirical evaluation. An Appendix is provided that details a “real” LMA that demonstrates the applicability of these contributions. Lastly, Chapter 6 concludes with closing remarks, and Chapter 7 presents a list of references.

Chapter 2 Background

This chapter presents the background information needed for this dissertation. It begins by reviewing forward chaining production systems (Section 2.1), and then introduces active databases (Section 2.2). The active database prototype used in this dissertation, VenusDB, is based on the fundamentals presented in this chapter.

2.1 Production Systems

2.1.1 Definitions

Forward chaining production systems consist of three components. These are a set of rules or *productions*, a representation of state called *working memory*, and an execution model implemented in the *inference engine*. Rules are composed of two parts: a *condition* and an *action*. The rule condition is a predicate over the working memory. Other common names for the rule condition are the *rule guard*, the *left hand side* (LHS), the *if part*, or the *subsequent*. The action of a rule is a list of statements or commands that modify the working memory. Other common names for the rule action are the *right hand side* (RHS), the *then part*, or the rule *consequent* [40,46].

The working memory is organized into groups of objects called *classes*. Classes are equivalent to database relations. Instances of a class are called *working memory elements* and are equivalent to database tuples.

2.1.2 Execution Semantics

The execution semantics of rule languages can be divided into two models: the OPS model [39,40] and the fixed point semantics model [35,67]. The OPS

model, developed for the OPS family of rule languages, is operationally determined by the behavior of the RETE *match algorithm* [39]. The OPS model proceeds by executing the *match-select-act* cycle. Rule evaluation begins in the match phase where the RETE algorithm evaluates all of the rule guards against working memory to produce a *conflict-set* of satisfied rules. The collection of individual working memory elements that satisfy a particular rule guard is called an *instantiation*. Next, the select phase is where the RETE algorithm picks a single instantiation from the conflict set to pass to the act phase. Lastly, the act phase is where the RETE algorithm executes the rule action, and the rule is said to have *fired*. This cycle continues until no further rules satisfy the match cycle.

The fixed point semantics model is derived from the formal semantics of Dijkstra's guarded command language [35]. Rule languages with formal semantics, such as UNITY, are most often used for program validation and/or parallel programming [67]. Fixed point semantics specify that a program may begin in any state satisfying a set of initial conditions. In each step of execution, a rule is selected nondeterministically and evaluated. This rule evaluation is *atomic*, a single transition in a state space. A restriction on the nondeterministic choice is that every rule is selected infinitely often, or in the colloquial, always has an the same chance of being selected. This restriction is called the *fairness* policy. Program execution continues until *fixed point* is reached, a state in which the execution of any rule does not alter the working memory.

The VenusDB language used in this dissertation is based on fixed point semantics. As such, VenusDB programs follow formally defined semantics where rule actions are defined to be atomic state transitions and programs terminate upon fixed point. These semantics allows formal reasoning about the program behavior and transaction models of VenusDB programs.

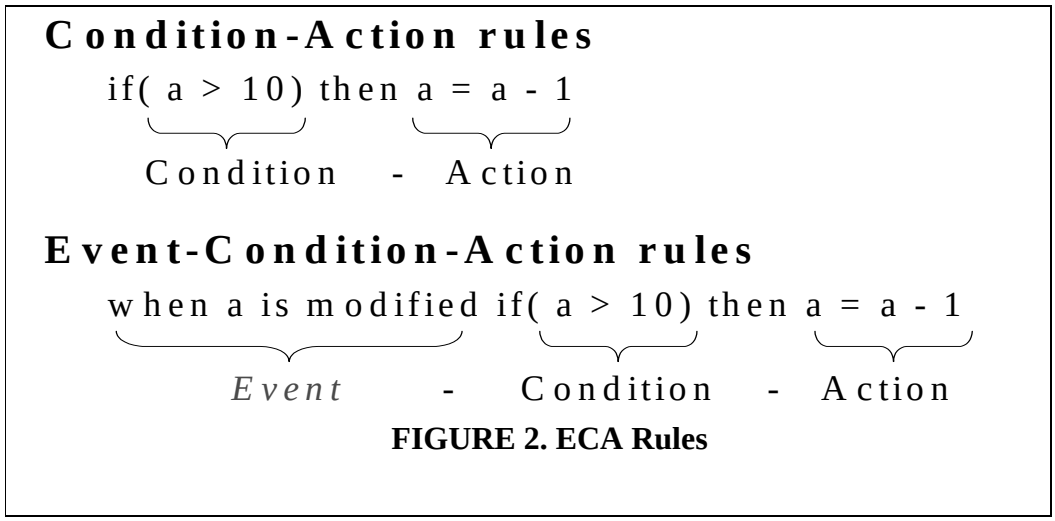


FIGURE 2. ECA Rules

2.2 Active Databases

Traditional databases are systems designed to durably store and efficiently search through large amounts of data. Active databases enhance traditional databases with the ability to react to database events. Active behavior is achieved by extending the database with three main components. First, active database programs are encoded as ECA rules. ECA rules are production rules that are evaluated only on specified events. Second, active databases address data durability and isolation issues in a series of coupling modes that relate rule execution and database transactions. Third, an architecture interface must be defined between production rules and the database facilities. This section presents these and other important design issues of active database systems.

2.2.1 ECA rules

Expert system rules are *Condition-Action rules* (CA-rules), rules evaluated on every update to working memory. In the active database paradigm, such evaluation is prohibitive since external processing spawns an unlimited number of

Primitive Events							
Event	**HiPAC**	**Ariel**	**POSTGRES**	**REACH**	**SAMOS**	**Sentinal**	**Starburst**
Database	√	√	√*	√	√	√	√
Transaction	√			√	√	√	
Temporal	√			√	√	√	
User Defined	√			√	√	√	

Composite Events							
Event	**HiPAC**	**Ariel**	**POSTGRES**	**REACH**[!]	**SAMOS**	**Sentinal**[†]	**Starburst**
Conjunction				√	√	√	
Disjunction	√			√	√	√	√
Negation				√	√		
Sequence	√			√	√	√	
Closure	√			√	√	√	
History				√	√		

* Additionally supports the retrieve event

[!] Additionally supports milestone events

[†] Additionally supports interval and periodic events

TABLE 1. Events Recognized by Active Database Systems

events to the database-stored working memory. Thus, active database rules follow the model proposed by HiPAC [33]. 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). Refer to Figure 2. ECA rules offer the programmer more flexibility than traditional CA rules, and they reduce the overall search space. For example, consider the rule in Figure 2 that maintains a counter with a maximum value of 10. The rule developer may have prior knowledge that this rule needs to be evaluated only when the counter is modified and will never execute in any other situation. ECA rules provide this flexibility.

The events recognized by ECA rules can be classified as *primitive events* or *composite events*. Primitive events are singular occurrences of events that can be subdivided into database events, transaction events, temporal events, and user-defined events [44,74,98]. Database events include the insertion, deletion, or updating of data within the database; transaction events include the begin, commit

		Event-Condition		
		immediate	deferred	decoupled
Condition-Action	immediate	condition checked and action executed in same transaction	not allowed	condition checked in action executed in separate transaction
	deferred	condition checked after event, action executed at end of transaction	condition checked and action executed at end of transaction	not allowed
	decoupled	condition checked after event, action executed in separate transaction	condition checked at the end of the transaction, action executed in separate transaction	condition checked in a separate transaction, action executed in another separate transaction

TABLE 2. Coupling Modes

or abort of a transaction; temporal events occur at absolute, relative or periodic time intervals; and user-defined events are signaled by the user application.

Composite events are described by a composition algebra. Besides the conjunction (evaluate if E1 & E2), disjunction (evaluate if E1 | E2), and negation of events (evaluate if E1 does not occur in an interval), composite events can be composed of a sequence (evaluate if E1 occurs before E2), closure (evaluate if E1 occurs one or more times in an interval), and history of events (evaluate if E1 occurs n times in an interval). Table 1 presents a summary of the events recognized by several active database prototypes.

2.2.2 Coupling Modes

The relationship between rule execution and database transactions has been addressed in a series of *coupling modes* [33,74,98]. 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* [33,74,98]. 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 commit; and in decoupled mode, execution of the pair occurs in separate transactions. Table 2 summarizes the semantics of these coupling modes. REACH proposes the additional modes of detached causally dependent in either of parallel, sequential or exclusive modes to give abort and commit semantics for decoupled transactions [21].

2.2.3 Concurrency Control and Recovery

A major difference between active databases and expert systems is that active database programs may contain readers and writers that are external to the rule programs. However, concurrency with respect to execution correctness between the rule application and other database users is ignored by most active database systems. This is because most active databases are designed to address "simple" rule systems. In this case, programs are (degenerately) correct if rules acquire the necessary locks [69,74]. However, correctness is not ensured if the active database programs consist of multiple rules. This dissertation addresses this deficiency.

The issue of recovery is similarly colored by a concentration on single rule transitions. It is assumed that the state prior to execution of the failed rule is a valid state. Therefore, a system rollback only recovers single rules. Other systems use the nested transaction model for rule recovery [33]. These systems still fail to consider recovery with respect to deep chains of rule firings and decoupled rules [74].

2.2.4 Architecture

The three most common active database architectures are *integrated*, *hybrid* and *layered*. Integrated approaches are implemented directly within the database. Advantages of this approach are that all database facilities are readily available, including the DBM's querying and transaction utilities. Thus, these systems are implemented without a loss in performance. An obvious disadvantage of this approach is that the database internals must be known. Consequently, a substantial amount of knowledge and effort is required for development.

The hybrid approach uses open database kits such as OpenOODB [96], Starburst [97], and Exodus [23]. This approach retains many of the advantages of integrated approaches, including performance, while reducing the development effort by publishing an internal database API. A disadvantage of the hybrid approach is that this published API tends to be quite complex. Further, applications developed using a hybrid architecture are still limited to a single database.

Last, the layered approach treats the database as a block box. Access to the database is only through standard database facilities (e.g. SQL queries). This approach has the advantages of providing active capabilities without any modification or knowledge of database internals. Further, in the presence of the SQL standard, the layered approach can provide active database capabilities to heterogeneous systems. This flexibility often comes at the expense of performance since substantial overhead may be incurred. This dissertation addresses this shortcoming by demonstrating the acceptable performance of a layered architecture within the VenusDB active database system.

	Relational			Object Oriented			
Architecture	**Ariel**	**POSTGRES**	**Starburst**	**HiPAC**	**REACH†**	**SAMOS**	**Sentinel**
integrated		√		√	√		√
hybrid	√		√				
layered					√	√	

† Re-implemented as an integrated architecture

TABLE 3. Architectural Design of Active Database Systems

Table 3 presents the architecture of several active databases. The table also presents the data representation (object-oriented or relational) of their respective databases.

2.2.5 Language Semantics

Active database language semantics are most often operational. As a result, many different semantics have been proposed. For example, Starburst defines behavior based on deferred coupling modes and delta relations, relations that store the net effect of database modifications within a transaction [8]. HiPAC defines semantics that allow developers to pick between immediate, deferred, and decoupled coupling modes within the nested transaction model. This nested behavior encourages concurrent rule execution [33]. Ariel, on the other hand, uses the TREAT match algorithm proposed for expert system languages [52,63].

Picouet and Vianu study this lack of formalism [78,79]. In their work, they develop a general framework for active database execution that is consistent with the intersection of the ARDL [82], HiPAC [33], Postgres [87], Starburst [97] and Sybase [88] technologies. Their resulting execution model operates in two phases. The first phase executes a queue of immediate coupling mode rules while updating both a queue of deferred coupling mode rules and the immediate queue. The first

phase ends when the immediate queue becomes empty. The second phase executes the rules in the deferred queue.

Other approaches formalize semantics by mapping active rules to deductive logic [81]. In this category, Zaniolo identifies a super-set of Datalog programs called XY-stratified programs [100]. Zaniolo then describes a transformation of an XY-stratified program to deductive logic. The transformed program, including extensional facts that represent database histories, executes using the formal models of deductive databases. Flesca and Greco propose semantics in which an active database program is transformed into a Datalog program [37]. The program is then executed by computing a stable model and updating the database [43]. Bidoit and Maabout perform a similar transformation as do Flesca and Greco [13]. However, Bidoit and Maabout's execution model is exactly the well-founded semantics of Datalog. A very different approach is described in [6]. Baral et al. propose the active database language, $\mathcal{L}_{active}$. $\mathcal{L}_{active}$'s syntax and semantics are derived from the causal action description language introduced in [42].

2.2.5.1 Confluence and Termination

Since rule evaluation is non-deterministic, another concern for active database language semantics is to determine properties that ensure program termination and *confluence*. A rule program is confluent when the termination state is independent of the order of rule execution. Termination is essential for real-time mission critical systems. Further, confluence is highly desirable or even critical in systems, such as financial management systems, where the termination state must be guaranteed, unambiguous, and reproducible.

Aiken et al. address termination and confluence in active databases by introducing static methods for analyzing rule programs [1,2]. Their algorithm proceeds by building a rule trigger graph from the input program. The graph is then

analyzed for cycles and commutative rules, pairs of rules that can execute in any order without influencing the trigger graph. The analysis either concludes that a program terminates and is confluent, or identifies suspect rules and give hints for fixing the problems. These methods are also used to determine observable determinism, the situation when actions viewable to the environment, like printing to a screen, are always appear the same regardless of rule execution order[1].

In addition to Aiken et al.'s work, many of the formalisms presented in Section 2.2.5 focus on the termination and confluence properties. Zaniolo introduced durable and ephemeral changes to the database that assist in deciding the termination problem [101]. Flesca et al. [37] describe how stable-model semantics ensure confluence for a larger class of problems than Aiken et al.'s static methods. Comai and Tanca exploit Datalog's well known confluence and termination properties by mapping active rules to Datalog [29]. Finally, Bailey et al. identify rule programs on the boundaries of (un)decidability [5].

2.2.6 Optimization

Active database program optimizations generally fall into two categories. The first category is the optimization of the underlying execution algorithms. As such, many active database prototypes [16,23,69] use the incremental algorithms used by the AI expert system languages such as the RETE [39], TREAT [63], and LEAPS [17] match algorithms. These algorithms are useful for optimizing rule programs that spawn deep chains of rule evaluation.

The second category of active database optimizations are within the rules themselves. Many of these optimizations are left to the underlying DBMS's query

1. Note, observable determinism and confluence are orthogonal: a confluent rule program is not necessarily observable deterministic and vice versa.

optimizers due to the predominance of the integrated architecture. However, there is a set of well excepted heuristics for optimizing active database rules. In [74], Paton presents some of these heuristics, including exploiting rule parameters and moving constraints as close to event evaluation as possible. Paton further explains how these heuristics allow multiple rules to be optimized using multiple query optimizers. His methods exploit the static methods for rule analysis presented in [1] to determine when it is possible to eliminate duplicate work.

Further optimization of multiple rules presented in the AI literature is often avoided [41]. This is largely due to the operational semantics of most active database languages. However, in [70], Obemeyer suggests a method for trigger filtering using decision trees. This method reduces the number of times predicates must be evaluated due to database events.

Chapter 5 of this dissertation expands upon these techniques to include methods for suggesting physical schema optimizations.

Chapter 3 The VenusDB Active Database Language

This chapter presents the active database prototype, VenusDB, that is used for the application work in this dissertation.

VenusDB is a platform for research in active database production systems. It is based on Venus, a main memory expert system shell designed to address the shortcomings of early rule languages. One of Venus' primary contributions is its introduction of structured programming within a rule paradigm. Venus rules are organized in parameterized modules whose semantics maintain the data driven execution of rule languages. Venus additionally addresses the performance problems of early rule languages by compiling its rules into C++ and executing the LEAPS match algorithm [16,17,64].

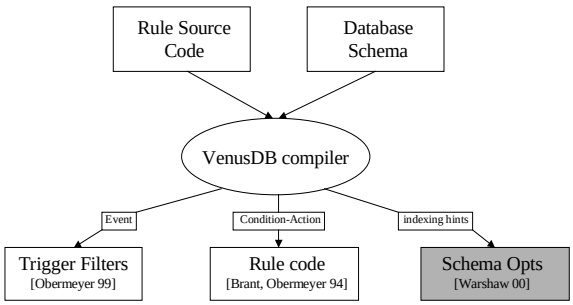


FIGURE 3. VenusDB Compiler Architecture

VenusDB is an extension of Venus that provides the benefits of the Venus language to active databases. Such extensions include the support for events. In effect, events move the Venus language from the Condition-Action rules of expert system languages to the Event-Condition-Action rules common to active databases. Other extensions include the installation of the Abstract Machine Interface (AMI), an API between the LEAPS match algorithm and persistent storage [69]. The AMI provides a tight integration between expert system behavior and active databases.

Due to its relationship to Venus, the VenusDB language is a compiled rule language. Its compiler was originally designed to take both rule language source and database schema information as input [30] (Figure 3). The output from the compiler was to be a set of trigger filters implementing the event mechanism [69], the C++ rule code that communicates with the LEAPS match algorithm, and a set of schema optimizations. This chapter describes the rule source, semantics, and extensions of Venus to the active database domain. The chapter concludes with a quantitative evaluation of the code complexity of programs written in VenusDB. The compiler-generated schema optimizations are covered in Chapter 5.

3.1 Venus Rule Language

3.1.1 C++ Heritage

The Venus language is syntactically modeled on C++, and retains C++ syntax wherever possible without introducing ambiguity or confusion. Data elements in Venus are defined as C++ classes and the data instances are C++ class instances.

3.1.2 Data

Venus supports the two data types of containers and primitive variables. Containers are set data, while primitive variables are individual variables.

Containers are denoted by square brackets ([]), intentionally drawing upon the C++ syntax for arrays. Container elements are accessed indirectly through cursors.

The cursors can be either existential and universal. The cursor type is selected by inserting a quantifier between the square brackets. An existential cursor is denoted by a question mark (?) quantifier, and corresponds to a positive con-

```
// for all u in r, if there does not exist
// a symmetric element e, then create and
// add it to the relation r
module enforce_symmetry(Relation r[])
{
     rule enforce;
     from r[?] e;
          r[*] u;
     if(!(u.domain() == e.range() &&
         (u.range() == e.domain())) {
          Relation i;
          i.setDomain() = e.range();
          i.setRange() = e.domain();
          r.insert(i);
     }
}
```

FIGURE 4. Example Module and Rule

dition element (e.g. `container[?]`). A universal cursor is denoted by an asterisk (*) quantifier. Universal quantifiers are always assumed to be within the scope of existential quantifiers. Consequently, the use of * usually corresponds to the use of negative condition elements in common expert system languages (e.g. `container[*]` is roughly equivalent to an OPS5 (`-container`)).

Primitive variables are similar to containers, except they always contain one element. A search is never required to find primitive variables. Therefore, their syntax is the same as C++ variables.

3.1.2.1 Rules

A Venus rule consists of a *header*, a *guard*, and an *action*. The combined guard and action is syntactically equivalent to a C++ `if` expression. The header specifies the rule name, an optional priority, and an optional from clause. The from clause is a syntactic shortcut borrowed from SQL. It allows the programmer to replace a container name and quantifier with a string. In Figure 4, the keyword

`from` and the asterisk declare `u` to be a universally quantified variable over the relation `r`. Similarly, the question mark declares `e` to be an existentially quantified variable.

A rule guard is a legal C++ boolean expression with restrictions. These restrictions constrain the use of externally defined C++ functions. These functions are allowed within the guard as long as they execute free from side effects and return a value.

Venus uses a subset of C++ as the action, or right hand side (RHS) language. The action is a list of updates, function calls, and Venus modules. However, an action cannot contain branching statements. The compiler parses the RHS and automatically recognizes updates to the state and inserts run-time calls notifying the inference engine.

3.1.3 Modules

The unit of organization in a Venus program is the module. A module consists of a formal parameter list, local variables, and rules. Venus adopted the C++ functional notation for module constructs, including curly braces to denote nesting program blocks. The scope of a module is limited to its actual parameters and local variables. To ease formal analysis, there are no global variables.

Figure 4 illustrates the code for a module called `enforce_symmetry`. This module is from a stylized presentation of a system developed from a specification of a device-structure hardware diagnosis program [31]. The module `enforce_symmetry` contains a rule which is parameterized over a type called `Relation`, that ensures that for all pairs (a,b) in the relation, a symmetric pair (b,a) is always present. In the application, this module is used to maintain lists of devices in a cluster that can serve as backups to each other. For example, if in the

cluster machine, a can backup machine b, then machine b can also backup machine a. By providing formal parameters to a module, a collection of rules can be "reused" for slightly different circumstances. By this it is meant that the module can be called once with one container as the actual parameter, and another time with a different container. This feature has been extensively used in all developed applications. Its use often leads to a significant reduction in the number of rules and the complexity of each rule [95].

A Venus module will fire rules until fixed point is reached. The entire RHS is treated as a single atomic action. Rules with the same priority, including multiple instantiations of the same rule, are selected for firing by a fair nondeterministic policy. For historical reasons, the current implementation dictates that fairness be defined such that a rule instantiation is fired at most *once*, where once has the OPS5 definition [39].

3.1.3.1 Module Semantics

A Venus program may consist of more than one module, and modules may be children of other modules. Module names and their actual parameters are listed in the action portion of a rule. Rules containing module calls in their actions are called *guard rules*. If a guard rule is satisfied, then rules in the embedded module may also fire. Modules may be embedded arbitrarily deep. The same module with the same or different actual parameters may be activated from multiple places in the code. At this time, the graph structure of the module hierarchy must form a static and directed acyclic graph (dag).

It is convenient but incorrect to think of a module as being called in the usual sense. An obvious firing of the guard rule is not necessary for execution of rules in the child module. If the guard rule is not satisfied, the module will not be called. If the guard rule is satisfied, the module *might* be called. In regard to mod-

ule calls, the fairness policy is extended. Whereas a normal rule will fire at most once on a single instantiation, a rule guarding a module call can potentially fire whenever there is a state change affecting the satisfaction of a rule predicate in the child module. Thus, the data driven nature of Venus extends through the rule guarding a module call. This explains the behavior of rules with guards of `if(TRUE)`.

Similarly, the predicate of a guard rule is not distributed over the rules in the guarded modules. As a consequence of the atomicity, even if the predicate is disabled during the execution of a child module, that module continues to fire rules until it reaches a fixed point. Again, though this behavior is derived from formal declarative definitions, the ultimate execution is consistent with procedural intuition. For example, in C++, when a function is called in the `then` portion of an `if` statement, the function will not stop if the guarding expression becomes false during execution.

Figure 5 illustrates the semantics of modules. Module `A` contains one formal parameter, `t,` which is a container over elements of type `T`. Initially, the rule `A::guard_C` will not fire since the local variable `x` is initialized to 0. As noted above, the evaluation of the `if(TRUE)` rule, `A::guard_B`, will automatically activate the `B` module with actual parameters `t` and `x`.

```
module B(T t[],
         int x){…}
module C(T t[]){…}
module A(T t[])
{
  int x = 0;

  rule guard_B;
  if(TRUE) {
    B(t,x);
  }
  rule guard_C;
  if(x==1) {
    C(t);
  }
}
```

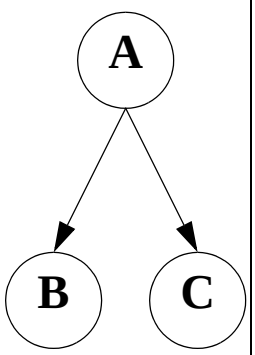


FIGURE 5. Module Definition and Resulting Call Graph

Should module B change the state of x such that upon reaching fixed point the value of x is 1, then control will pass to module C. Should module C modify the t container, then control will pass to module B after module C reaches fixed point. Execution will continue in this manner, bouncing back and forth between the B and C modules, until A reaches fixed point.

3.1.4 Polymorphism

Venus is designed to inference over data. Venus supports polymorphism with respect to this data in two ways.

First, Venus supports a formalized interface between data stores and the inference engine via the Abstract Machine Interface [69]. The formal parameters to a Venus module only specify the type of the object that resides within a container. It does not specify the underlying container implementation. Thus, successive calls to a module may have as actual parameters different containers with different implementations. For example, the module defined in Figure 4 specifies as its formal parameter a container consisting of objects of type `Relation`. Consider two separate invocations of the `enforce_symmetry` module. The first call to the module might be with a container implemented as a main memory doubly linked list. The second call to the container might be with a container implemented as an Oracle table. In the first case, a call to `r.insert(i)` results in inserting a node into the list, whereas in the second case, it results in adding a tuple into the Oracle database.

Second, Venus supports inheritance within the C++ data stored in containers. In the example rule in Figure 4, the container `r` consists of elements of type `Relation`. The accessor methods `domain()` and `range()` return the object ids of elements. If these methods are defined as C++ virtual methods, then a container of elements inheriting from type `Relation` will be passed to Venus as an

actual parameter, and the method calls will be dynamically dispatched to the appropriate derived class.

3.2 VenusDB Modifications

The main differences between Venus and VenusDB are the inclusion of active database events and an interface between the LEAPS match and persistent store called the AMI.

3.2.1 Events

The inclusion of events in the VenusDB language makes it an ECA language, as opposed to the Venus language, which is CA.

The VenusDB version of events is different than that of most other active database languages. This is due to VenusDB's aim of supporting complex data driven applications rather than system services and lower level applications.

3.2.1.1 Database Events

VenusDB supports the primitive database events insert, update, and remove. This allows a programmer to customize the behavior of rules based on the triggering event. Different VenusDB data types are sensitive to different database events. This is generally consistent with other active database prototype efforts.

Event declarations are optional and rule scoped. By default, data types within rules that do not specify events are monitored for all applicable events. Table 4 presents the events recognized by the VenusDB data types. A full discussion of VenusDB primitive events is presented in [69.

	Event†		
Data Type	**insert**	**update**	**remove**
Primitive Variable		√	
Existential Cursor	√	√	
Universal Cursor		√	√
Container	√	√	√

† The default event, all, is the conjunction of the checked events for each data type

TABLE 4. VenusDB Data Types and

3.2.1.2 Transaction Events

VenusDB does not support transaction events (e.g. begin, commit, and abort). This is because of VenusDB's focus on application programs, not system services. At the application level, transaction rollbacks typically only occur at the periphery of the system (e.g. at a data entry station). Once entered into the application's data flow, data is permanent and therefore already committed. It follows that it is sufficient for VenusDB to support database events, and thus, triggering on transaction events is not necessary.

3.2.1.3 Temporal Events

Fine grain temporal events are not supported. Course grain temporal events are indirectly supported. This is by activating the rule system through a cron or equivalent system clock service. This is not explicitly recognized in the language. Integrating temporal events into VenusDB is an open issue.

3.2.1.4 Composite Events

Composite events in the style of HiPAC are not supported. However, it is important to note the VenusDB does support an implicit disjunction over the database events listed in a rule's from clause. This preserves the data driven behavior of VenusDB.

3.2.2 Abstract Machine Interface

VenusDB operates by issuing commands to the Abstract Machine Interface (AMI), an instruction set used by the Venus run-time match algorithm, LEAPS. The purpose of the AMI is to form a tight integration within a layered architecture. The AMI is the only system to publish such a high-level interface between the rule and database portions [66,69].

VenusDB is implemented as a layered architecture for two reasons. First, VenusDB is a heterogeneous rule engine. Thus, the hybrid or integrated approaches are impractical since they would require a knowledge of detailed information about access and implementations of multiple (commercial) database systems. Second, only the layered approach is portable satisfying one of the original design goals of the VenusDB language. The research presented in this dissertation provides significant evidence that the layered approach is feasible within real systems that solve real problems.

The AMI is defined by a set of abstract C++ classes. An implementation of the AMI for a particular database is four C++ classes that inherit from the appropriate AMI base classes. These implementations consist of two container implementation classes and two cursor implementation classes. AMI implementation classes use the C++ `template` facility to provide type safety. This C++ interface limits VenusDB to databases that support a C or C++ database access. With the wide acceptance of the ODBC standard, nearly all databases now support such an interface. A full description of the AMI is presented in [69].

3.2.2.1 AMI Optimizations

Of particular interest to this research are the AMI's optimization utilities. These are primarily accomplished by leveraging the advanced query facilities of composite databases when available. The VenusDB compiler identifies predicates that are applied to a cursor in a particular rule. These predicates are pushed down through the AMI to the local database layer for query processing.

The AMI's implementation of this facility is quite flexible. If the component database supports an advanced query capability, the predicates will be executed on the database and may result in a significant performance benefit. If the component has no query support, the predicate is ignored and a full relation scan occurs.

The definition and implementation of the AMI predicate facility is being refined and heavily exploited by this research. Chapter 5 explains this utility.

3.2.3 Concurrency Control

Concurrency control issues are traditionally managed by using coupling modes and the underlying concurrency mechanisms of the database as presented in Section 2.2. VenusDB does not address such issues. It is the author's belief that coupling modes are unmanageable in programs that contain many rules that may chain. In other words, coupling modes place undue burden upon the programmer on the very programs for which VenusDB is designed.

Correl and Miranker, however, propose a concurrency control scheme based upon the modularity features of VenusDB [30]. This scheme attaches isolation specifications to individual modules. Three categories of data isolation modes are proposed called *guard stability, serializable*, and *exclusive*. Guard stability (modeled after cursor stability) allows the greatest amount of concurrency, but pro-

vides the least amount of isolation from other users. This mode dictates that, at minimum, a row 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. For this reason, this concurrency control model has not been adapted into the VenusDB language. This dissertation poses the hypothesis that concurrency models are application dependent. As such, it begins the investigation of the feasibility of modifying the VenusDB compiler to take as input a rule program and problem type and output transformed code with the appropriate concurrency model (Chapter 4).

3.3 VenusDB Language Semantics: An Evaluation

At its core, VenusDB is an instance of a forward-chaining rule language. Such languages are in common use as a method of knowledge representation and the basis of expert-system programs [20]. The paradigm has also gained notoriety by serving as the basis of systems that are difficult and expensive to maintain [62,65,83]. A specific power of this representation is that it is data driven and on each cycle of execution, any one of a large number of problem solving alternatives may be selected. Yet, each alternative can be expressed in isolation as a single rule. This capability is both a blessing and a curse [9,53]. Though critical to the success of the paradigm, this flat monolithic architecture has proven to be very difficult to maintain. Given a seemingly local update to a large rule program, the global scope of a rule's condition regularly introduces bugs elsewhere in the program.

More recently, the application of forward-chaining rule languages to active databases have compounded the problems of the paradigm due to concurrent execution of other transactions. In other words, it is precisely the ability to prescribe actions to be taken based on a partial definition of a state and/or events on the database, which are independent of the source of an update, that has led to active databases. It is also this ability that leads to difficult issues in semantics and correctness of active database programs (Section 2.2.5). Further, active database languages are usually monolithic since the initial focus of the languages was on implementing core system services (such as view maintenance and integrity constraints). In such programs, rules are essentially independent programs in which there truly is no relationship between rules. In addition, SQL is inherently monolithic. Many active database languages extend SQL, syntactically or conceptually, and therefore, they inherit SQL's lack of program organization.

VenusDB addresses these semantic issues through a formal language definition for structured rule programing. Section 3.1 explains how VenusDB's module semantics lead simultaneously to the evaluation of every rule on every cycle, yet introduce structure limiting spurious interactions among rules. We believe that VenusDB's definition for structured programming improves upon the alternative solutions that have been proposed (which are often procedural in nature) [7,62]. Section 3.3.1 elaborates on these alternatives.

This section, 3.3, presents an evaluation of VenusDB's language semantics as they relate to code complexity. The study is performed by comparing an OPS5 implementation of a deployed expert-system, ALEXSYS, with an implementation in Venus called REALESYS. ALEXSYS proved ideal for this study for several reasons. First and foremost, a version of ALEXSYS is deployed and the prototype was readily available. Originally developed for Citicorp, the program is now in widespread use. Secondly, though the program is of moderate size, 44 rules, it's

roughly the complexity of a “real world” example of a hard expert system application, the class of applications addressed in this dissertation[1].

3.3.1 Related Work

Identifying the lack of hierarchical decomposition in rule-based programs as a culprit in the life-cycle costs of rule programs is not new. In fact, the problem comes up almost immediately in any large scale effort [4]. Following is a taxonomy of solutions.

- Structural Protocols: Within a flat monolithic rule language, such as OPS5, specific protocols concerning the structure of rules have been suggested [4,75].

- Rule Groups: In a number of systems rules can be grouped together into “rule groups’’. The flow of control among the groups is performed by explicit procedural invocation [38,45,53]. In [7], a method is proposed in which rule developers must establish metrics to determine the stratum classification of rules. Program execution proceeds by moving from lower priority stratums to higher priority stratums. Other approaches include the use of regular expressions in a meta-level to define legal sequences [45,49].

- Object Embedded: Rule groups are defined as objects and/or rules that are used to define individual members of an object [14,65,73].

An aspect shared by each of the above solutions is that they are procedural in nature. For example, in most of the rule group solutions, control moves from module to module as the result of a jump, or a jump to subroutine, executed in the action of a rule. Consequently, only one module at a time is sensitive to the current

1. Hard active database technology been available at the time of the program’s development, the technology would have been exploited [85]

```
rule fill_a_bin_rules;        //module one, fill a bin
if(goal.context == fill_a_bin)//pattern/message
     {...}                    //signaling module one

rule finished_filling_a_bins;
if(goal.context == fill_a_bin &&
   ...)      //recognize termination of the module
     {goal.context = fill_b_bin;} //change goal

rule fill_b_bin_rules;   //module two, fill b bin
if(goal.context == fill_b_bin)
{...}
```

FIGURE 6. Modularity by virtue of a "Secret-Message"

state of the program. Thus, not all possible alternatives are evaluated at each cycle. The correctness of the program requires the rule developer to consider both data-driven and procedural methods of programing.

The motivations behind VenusDB are most similar to those that produced the development of the RIME protocol for R1/XCON [4]. Also, it was only through the recognition of these protocols that the modular structure for the OPS5 ALEXSYS could be inferred.

To illustrate the structural protocol, consider an abstraction of the control mechanism used in the ALEXSYS program presented in Figure 6. The code fragment illustrates the notion of *secret messages* to mimic encapsulation [62]. Inspection reveals that every rule contains a predicate on an element of type goal. Rules exploit *goal elements* in order to form rule groups. The presence (/absence) of a particular goal element in working memory enables (/disables) rule groups. Rules detecting the termination of a subgoal (e.g. rule `finished_filling_a_bins`) change the value of the goal element moving control to the next group. In essence, this protocol is synonymous with a nested transaction model where rules sharing a common goal pattern are grouped within a nested

transaction. Rules that change goal elements commit transactions and initiate a new nested transaction. Program execution operates by moving between rule groups (transactions) and executing to completion (fixed-point). This form of hierarchical programming has been coined "secret messaging" due to the fact that unless a programmer is already familiar with the practice, there is nothing to distinguish these special goal patterns as control patterns.

The structural protocols do have the desirable feature of managing control-flow in a strictly data-driven fashion. Rigid adherence to the protocols eliminates undesirable *cross-talk* among the rule groups. The protocols manifest groups of related rules and key transitions in control flow. Operationally, it has been reported that reusing templates has some of the benefits of code reuse. That is, basic coding structures will not have to be rewritten from scratch each time they are needed, leading to reduced development and maintenance costs [4].

A problem with the structural protocols is that they are not part of the underlying language. Thus, syntactic checking of application source code for adherence to the protocol, if done at all, must be conducted by a separate compiler. By analogy, this is identical to using the program verifier `lint` with C programs [55]. Other problems with structural protocols, that have been addressed in VenusDB, include the duplication of rules in a template rather than parameterizing a single copy to implement similar tasks. As a result, a simple maintenance change may involve ensuring that several sections of code are modified. Lastly, despite exploiting the data-driven execution model, the use of goal elements is still fundamentally procedural and limits the sensitivity of rules.

A Venus re-implementation of the OPS5 fragment of Figure 6 is illustrated in Figure 7. In this figure, control information is explicitly stated in a separate module. The appearance of control patterns as secret messages is eliminated, simplifying (compared to the OPS5) the guards of the rules in sub-modules. Transaction issues are also explicitly isolated by VenusDB modules easing the integration within a database. Additionally, rule priorities present a straightforward manner to further control rule firings.

```
module load_bins (log_type logs[]
{
 rule fill_a_bins    priority 10;
 if(TRUE)
     do_fill_a(logs[]);

 rule fill_b_bins    priority 9;
 if(TRUE)
     do_fill_b(logs[]);
}
```

FIGURE 7. The Venus Method for Procedural Control Expressed by the OPS5 in Figure 6

Through implementations such as the one in Figure 7, VenusDB language semantics encourage passive programming paradigms consistent with the methodology of stepwise refinement, a paradigm that is often used for formal methods of program development. Stepwise refinement paradigms contain initial pseudo-code statements that are elaborated step-by-step into final executable code, much like top-down designs [84]. The design of REALESYS exploits stepwise refinement.

3.3.2 The Mortgage Pool Allocation Problem

ALEXSYS and REALESYS implement a solution to the mortgage pool allocation problem which is faced several times a month by financial institutions who serve as market-makers in mortgage-backed securities. The problem is for the market-makers to map a collection of buy orders for mortgage-backed securities (e.g. Fannie-Mae) to a collection of sell orders. It is the market-makers who

arrange the tangible details concerning the transfer of the security and the exchange of money. The market-makers maintain a small inventory of securities to facilitate their work. This inventory is organized into *pools*. A request to purchase securities is termed a *contract*. Contracts and pools may be of different sizes. When settling a trade, a pool from inventory may be cut into many pieces to fulfill one or more contracts. Similarly, more then one pool may be used to fill a contract. However, it is insufficient for settlements to be arithmetically correct. The Public Securities Administration has published many pages of detailed constraints limiting how pools are assigned to buy orders [84].

3.3.2.1 ALEXSYS

The ALocation EXpert SYStem (ALEXSYS) is an OPS5 implementation of a solution to the mortgage pool allocation problem. ALEXSYS exploits eight different heuristic method/regulatory combinations to define and fulfill buy orders. After deciphering the organization of the secret messages in ALEXSYS, it is possible to expose the modular structure developed by the original authors. The 44 rules of ALEXSYS form 12 modules whose control flow is illustrated in Figure 8**.**

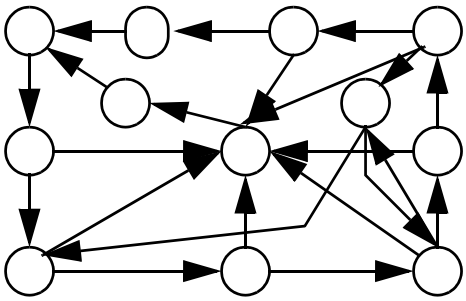


FIGURE 8. ALEXSYS Call Graph

The longest acyclic path through the call graph is of depth nine. Yet, the graph is made up of many cycles. In fact, the starting node of the graph is ambiguous as drawn and only slightly clearer when reading the OPS5 source code. This cyclic graph makes it is difficult to determine how control reaches a node/rule to become active. Further, the cycles present in the OPS5 version were directly responsible for making debugging the translation of ALEXSYS into VenusDB a

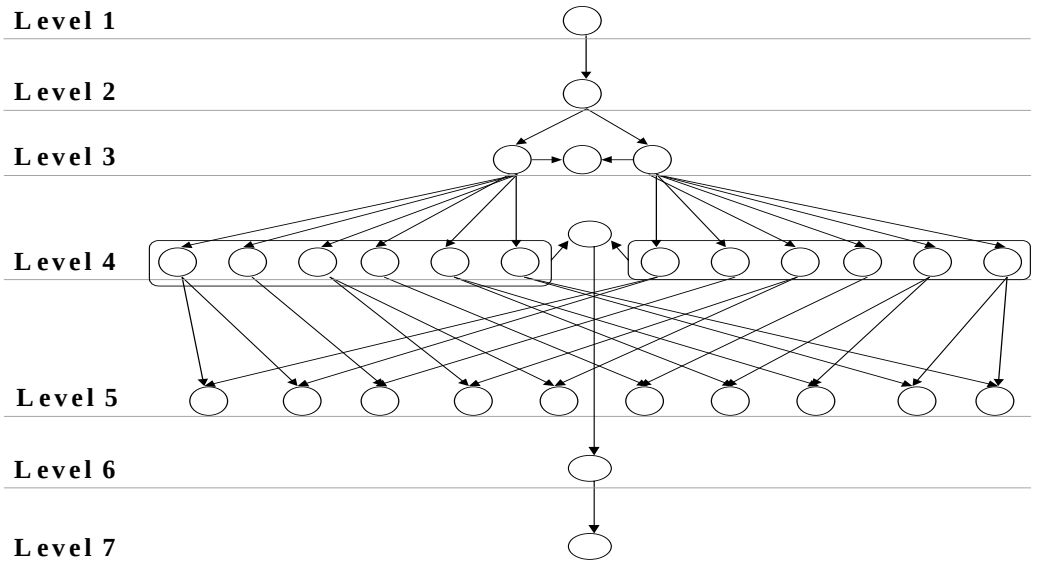


FIGURE 9. REALESYS Call Graph

challenge. For comparison purposes in this case study, the structural protocol used by ALEXSYS will be considered equivalent to the VenusDB modules.

3.3.2.2 REALESYS

The stepwise Refined ALlocation Expert SYStem (REALESYS) is the VenusDB implementation of a solution to the mortgage pool allocation problem. As the name implies, REALESYS is implemented using a top-down design with stepwise refinement. For purposes of this study, REALESYS uses the same basic collection of greedy heuristic methods as ALEXSYS to achieve as close as possible execution results for accurate comparisons. Figure 9 illustrates the call graph of REALESYS.

The longest acyclic path within REALESYS is of depth 7. Vertices with an out-degree greater than 0 represent parent modules (i.e. modules with guard rules within them). Orthogonally, vertices with an in-degree greater than 0 represent code reused modules (which is not possible in the OPS5 implementation). For

illustration purposes, all of the nodes within the large boxes in Level 4 call the module pointed to by the boxes (an area of particularly interesting code re-use). Inspection easily reveals the root node as the top-most node. Cycles are nonexistent. Thus, the sequence of state transitions that reach a particular rule firing is clear in the REALASYS program. This is in sharp contrast with the cyclic ALEXSYS call graph.

We did experience a significant growth in the total number of rules. ALEXSYS contains 44 rules while REALESYS contains 74. However, 28 of the 74 rules in REALESYS are of the form "if(TRUE)". Such rules are used primarily as control constructs which do not test any of the current state of execution. Thus, only 46 rules within REALESYS contain rules with predicates. Even eliminating the "if(TRUE)" rules, and despite code reuse in the form of parameterized modules, the number of rules surprisingly increased. Upon inspection, it was determined that it is frequently possible to align a single OPS5 rule with a number of VenusDB rules. As is often the case, the single OPS5 rule contains both control and problem-state conditions, while the VenusDB representation splits these elements into parts. Nevertheless, the quantitative analysis presented in Section 3.3.3 demonstrates that by all other measures the code is simpler.

The length of the longest acyclic path in REALESYS (7) corresponds directly to seven levels of stepwise refinement designed for the REALESYS implementation. Each level divides the program organization in such a way as to solve a step of the program while simplifying the guards of the next refinement steps. The levels perform the following functions:

Level 1. Handles control for incoming trades.

Level 2. Determines if profit is to be gained by filling a contract +/- 2.5% of its value (the allowable range to fulfill a contract).

Level 3. Calls to different rank ordered heuristics to fulfill a contract.

Level 4. Filters on sufficient conditions for the application of the heuristic.

Level 5. Filters on necessary conditions, determined by government regulations for the application of the heuristic.

Level 6. Reports a successfully filled contract.

Level 7. Reports the details of the trade.

By virtue of this organization, all significant control information has been narrowed to Level 3 and stated explicitly. Further, the testing for correctness and ranking quality of the heuristics within REALESYS is limited to manipulating the rule priorities of the two third level modules. Lastly, independent of the sufficient conditions for a heuristic, which appear in different rules, the satisfaction of government regulations is localized. Thus, changes in government regulations will likely result is small changes to REALESYS. This is not the case for ALEXSYS.

3.3.3 Quantitative Results

The ideal test for design and long-term maintenance of the two implementations would be to deploy both independently, and to carefully account for manpower costs, number of bugs, average time to repair a bug, etc. Such large scale testing is impractical and has rarely been performed. However, a number of efforts have resulted in quantitative software metrics that have, in smaller scale studies, been correlated to the life-cycle costs of computer software [11,61].

Any one software metric will not give a full picture of a program's quality. As a result, software metrics have evolved to cover three basic measurement domains. These domains span the areas of 1) *volume* - size of the program, 2)

control flow - complexity of the execution paths, and 3) *information flow* - complexity of the data flow [49]. Four different software metrics were chosen to measure ALEXSYS vs. REALESYS covering each of these three domains. The metrics are conditions per rule, lines of code, McCabe's cyclomatic complexity [62], and fan-out[2] [53]. Conditions per rule and lines of code are straight-forward volume metrics. McCabe's cyclomatic complexity, developed by Thomas McCabe, is a control flow metric that measures the number of paths through a program. Fan-out[2] is an information flow metric that analyzes the read and write sets within modules. Our results on information flow are nearly identical to the other three measures and are therefore omitted. All of these metrics have been shown to correlate to the development and life-cycle costs of software [50].

3.3.3.1 Conditions Per Rule

The volume metric conditions per rule give evidence of the guard complexity of a rule-based program. For an OPS5 rule, the number of conditions is defined as the number of features in a rule per the OPS5 resolution strategy. For a VenusDB rule the equivalent measure is defined as the number of AND and OR connectives separating relational tests plus 1, plus 1 more if a priority is mentioned[2].

Conditions per rule is considered by many rule-based programmers as the most telling metric of the complexity of a rule-based program [50]. Many conditions per rule imply complex guards with a large potential for error. Few conditions per rule, in contrast, represent less complex guards with a small potential for error.

2. Priority mechanisms can be simulated through a single additional condition and a penalty of 1 seems appropriate.

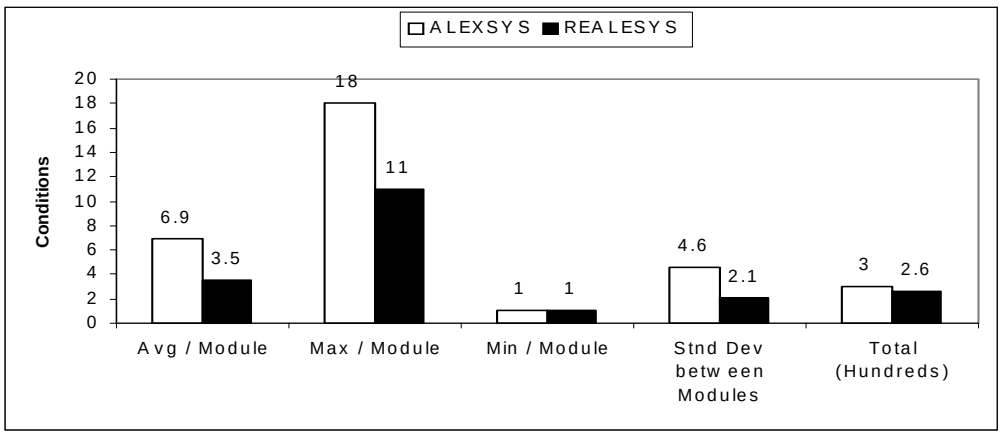


FIGURE 10. Conditions Per Rule

The results of Figure 10 are positive for REALESYS. REALESYS yields roughly half the average conditions per rule and half the standard deviation as compared to ALEXSYS. The maximum size rule in REALESYS is about 40% smaller than the maximum size rule in ALEXSYS. Interestingly, REALESYS contains 43 fewer conditions than ALEXSYS. This reduction in total conditions can be primarily attributed to the code reusable modules in the graph of Figure 9 combined with the elimination of secret messages by using VenusDB's modularity. Thus, REALESYS contains 14% less conditions than ALEXSYS to get the same results.

3.3.3.2 Lines of Code

Lines of code is a volume metric that is frequently given for iterative programs as a measure of the overall complexity of a program. The more lines of code any program may have, the more likely it is to contain typographical errors, code repetition, and complex statements.

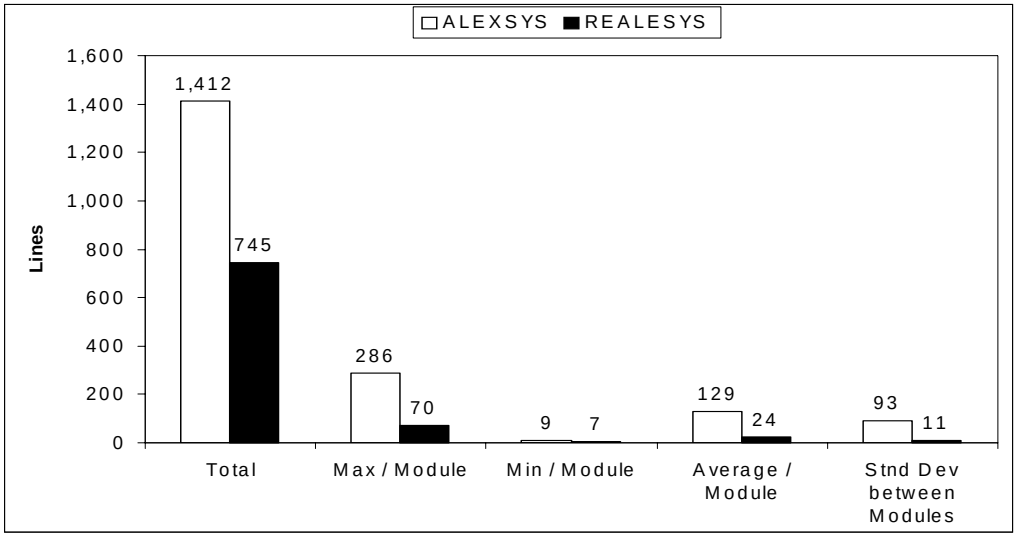


FIGURE 11. Lines of Code

One line of code is defined as any non-comment line with code, a module call, or a function call.

The results of Figure 11 continue to demonstrate about a twofold complexity improvement of REALESYS as compared to ALEXSYS. The results also demonstrate that the standard deviation between modules is about nine times greater for ALEXSYS. The fewer lines of code and smaller standard deviation for REALESYS supports the claim that though REALESYS has about three times as many modules as ALEXSYS, the modules tend to be much smaller than those of ALEXSYS with less complex rules. In other words, the additional modules in REALESYS actually reduce complexity rather than increase complexity. This claim is further supported with McCabe's cyclomatic complexity and fan-out[2].

3.3.3.3 McCabe's Cyclomatic Complexity

Cyclomatic complexity measures the complexity of control flow by extracting the number of possible paths through a program. Cyclomatic complex-

```
void module(/* parameters */)
while(!fixed_point)
     switch(non_deterministic_selection(i))
     {
     case 1:  {
          rule LHS_Test;
          if( /* LHS Test */ )
          {  atomic RHS action; }
     case 2:  {
          rule IF_TRUE_rule;
          /* if(TRUE) has no test */
          {  atomic RHS action;}

     }}}
```

FIGURE 12. Abstract Execution Structure for the Application of Cyclomatic Complexity to Rule Systems

ity, $v(G)$, of a program graph of n vertices, e edges, and p connected components is calculated by:

$$v(G) = e - n + 2 * p \quad \text{(EQ 1)}$$

or

$$v(G) = \text{number of decision statements} + 1 \quad \text{(EQ 2)}$$

In equation 2, a decision statement, called a predicate, is any conditional branch within a module [61]. For iterative programs, a cyclomatic complexity of less than or equal to 11 for a given module is considered acceptable. Since the control of rule-based programs differ from iterative programs, cyclomatic complexity must be adapted to rule-based programming. This is done conceptually by mapping VenusDB modules to the procedural structure as illustrated in Figure 12.[3]

3. This is different than Pasik's mapping which requires first deciphering secret messages[76].

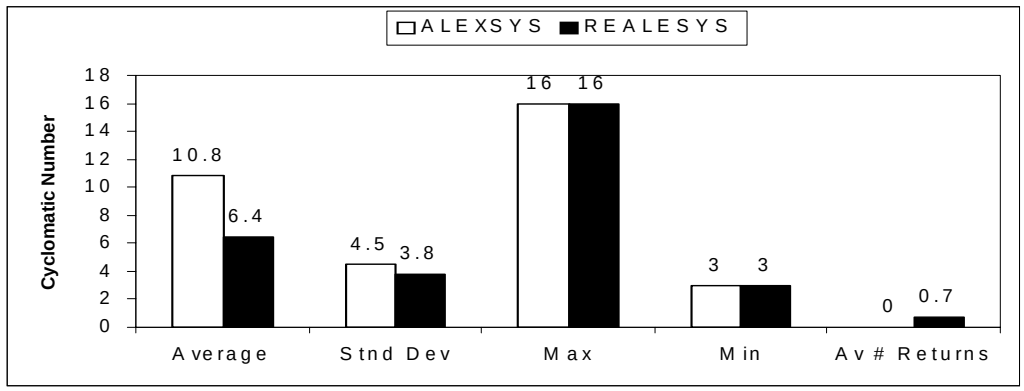


FIGURE 13. McCabe's Cyclomatic Complexity per Module

The results of Figure 13 show an overall cyclomatic complexity improvement of about 41% (continuing the roughly twofold complexity improvement of REALESYS as compared to ALEXSYS) and a slightly smaller standard deviation for REALESYS. This improvement suggests that REALESYS simplifies modules by dividing them into parts. Additionally, this demonstrates the effectiveness of top-down design with stepwise refinement. It was expected that the minimum complexity rules would be similar for both REALESYS and ALEXSYS; several small rules are typical of rule-based systems [50].

Surprisingly, both REALESYS and ALEXSYS have an equivalent value of 16 for their most complex module. Inspection reveals that these modules do not serve the same function. The module of complexity 16 within REALESYS stems from the control modules, Level 3 in Figure 9. These modules are composed of a series of eight "if(TRUE)" rules, each with a specified priority. It seems harsh to penalize what expresses, in essence, a simple sequence of subroutine calls whose direct translation into C++ would reduce the cyclomatic value by half. The use of priorities in this fashion is a recognized failure of VenusDB. Priorities are a procedural construct that improve upon structured protocols. In this respect, a comparison to the OPS5 version reveals that if all of the control rules in ALEXSYS were

aligned into a single module, the cyclomatic complexity for this module would be roughly twice that of the REALESYS implementation.

3.3.4 Discussion and Conclusions

Is this a controlled experiment? As above, a completely controlled experiment would involve independent programming teams over several years. Within reasonable confines, we believe we have done a reasonable study. This is not an occasion where a second implementation is simply better than the first. Experienced knowledge engineers threw away at least two earlier versions of ALEXSYS. In this experiment, two implementations of ALEXSYS written in VenusDB were produced (a direct naive translation of ALEXSYS, and REALESYS). The naive translation of ALEXSYS shrank the total number of lines of code by about 10%. There were no difference in any other measures. Thus, the possibility that the improvements were due to superficial syntax can be ruled out, and it can be concluded that these improvements are indeed due to the language constructs intended to support structured programming. Note that it is *not possible* to translate REALESYS back to OPS5; REALESYS exploits code reuse through parameterized modules. Even so, a mapping back to OPS5 would, like RIME, rely on programmers to follow a protocol rather than making the elements of structured programming part of the language and checkable by the compiler.

Despite an increase in the number of rules, VenusDB provides significant improvements over OPS5 in all other metrics in the encoding of the mortgage pool allocation problem. REALESYS displays roughly a twofold complexity simplification in each of the four metrics spanning the three basic measurement domains of software metrics [49]. These measures have been accepted as measures of program quality and have been shown to correlate with long-term development and maintenance costs. Thus, it can be concluded that the method of encapsulation

developed for VenusDB represents an important step toward reducing the engineering costs of expert-system and active database programs.

Chapter 4 Application Semantics for Active Log Monitoring Applications

Active database applications are not simply production systems applied to data within a database; rule computation must be integrated within the database's transaction model. The most widely accepted approach is for active database developers to relate rule processing to database transactions through a pair of *coupling modes* [33] (Section 2.2.2). The flexible specification of events and coupling modes is intended to maximize system throughput [34]. However, the progression of research has lead to the development of dozens of coupling modes [21,24,33]. As a result, coupling modes often burden application programmers with difficult conceptual specifications. This complexity becomes virtually unmanageable within hard active database applications where hundreds of rules may interact. This chapter begins deciphering which coupling modes are necessary to achieve useful active database programming.

In the development and study of a number of rule programs, the author has observed that many of these systems can be classified into a subclass of hard rule systems coined *Log Monitoring Applications* (LMAs). LMAs are expert system applications that analyze logs maintained in a database. Applications within the LMA class range across point of sale, medical patient, network security monitors, real-time decision control systems, and process control monitors [18,31,85,93,94, 95][1]. In each of these applications, a database is chosen as a storage medium due to commercial DBMS's query and data durability services that are exploited as a platform for post-processing, analysis, and decision-support. A fundamental property of an LMA is that its logs are only appended to, but data within the logs are

1. It is the author's opinion that many other applications that do not intrinsically satisfy the LMA restrictions may still be implemented as an LMA.

never updated or deleted. For example, consider a network security monitor that analyzes network traffic in order to detect computer hackers (Appendix). In this application, network traffic and suspicious packets are logged to a database. These logs can never be modified; they represent network logs that must be maintained for forensics. Thus, an LMA implementation uses active rules to monitor network logs as traffic and interesting packets are inserted into the database. A contribution of this research is to show how this write-once nature of the logs can be exploited to insulate application programmers from the complexity of using coupling modes. This chapter presents a formal study, using active constructions, of the resulting simplifications that can be made of active database programs that obey the LMA restrictions.

The theory of this chapter is applied within a general framework so that it is applicable to a wide scope of applications. Therefore, the chapter concludes by explaining how the LMA properties can be exploited within the VenusDB platform (Section 4.10).

4.1 Motivation

The current methods of integrating rules into a database add complexity to the development of active database systems. In part, this complexity is introduced to reduce the duration and number of locks taken by rule execution. Rule execution is often a long-running activity (Chapter 5). The resulting long-duration locks may significantly reduce overall system performance. Dayal et al. summarize this situation in their 1990 SIGMOD paper by saying that “executing a long-running activity as a single transaction is not strictly necessary in most cases, and can significantly delay the execution of short transactions” [34].

As a consequence of the long-duration locks taken by active rules, the relationship between rules and database transactions has been addressed in a series of *coupling modes* [33,34,74,98]. Coupling modes provide application developers with a flexible mechanism for rule integration. However as this section demonstrates, this added flexibility often leads to quite different program semantics. These semantic differences prove to be an obstacle when developing the large quantities of rules that embody hard active database applications. A goal of the research presented in this chapter is to insulate application programmers from the complexity added by coupling modes while maintaining their performance benefits. This chapter accomplishes this goal by presenting formal proofs of concurrency schemes for LMAs using coupling modes. The constructive nature of these proofs can be exploited by compiler-based systems.

4.1.1 Coupling Modes

Traditional expert-system and guarded command rules are *Condition-Action rules* (CA-rules), rules evaluated on every update to the database. In the active database paradigm, such evaluation is prohibitive due to its multi-user environment. Therefore, active databases extend rules to include an *event* clause. The event clause in an algebraic expression that describes the occurring event(s) that must occur in order to evaluate a rule. The resulting rules are called *Event-Condition-Action rules* (ECA-rules).

Coupling modes map ECA rules to transactions. There are 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 most predominately used coupling modes (and the ones used in this chapter) are *immediate*, *deferred*, and *decoupled* [33,74,98]. 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 commit; and in decoupled mode, execution of the pair occurs in separate transactions. Table 2 in Section 2.2.1 summarizes the semantics of these coupling modes.

4.1.2 Example 1

Figure 14 illustrates the complexities introduced to application developers by coupling modes. The formal notation illustrated in the figure is presented in Section 4.4. Following is its informal description.

Figure 14 contains a single rule, *R*, and two tables, T_0 and T_1. The event clause of *R* monitors for insertions into T_0. In other words, insertions into T_0 trigger *R* for rule evaluation. *R*'s conditions clause is satisfied when there does not exist the value 3 within T_0. The action of *R* inserts the value of 2 into T_1. Let the initial state of the database be empty, i.e., tables T_0 and T_1 are empty. This is denoted by $\mathcal{D}_0 = \{\}$. Rule processing begins due to the execution of *external events*, modifications to the database that occur from sources outside of the processing of rules. In this example, two external events are executed in sequence. The first external event, denoted by $\mathcal{X}_0$, inserts the value 1 into T_0. The second external event, denoted by $\mathcal{X}_1$, inserts the value 3 into T_0. Following is an explanation of two different execution scenarios that can occur when executing *R* using the coupling mode assignments of E-C and C-A immediate modes versus E-C and C-A decoupled modes.

rule R $\mathcal{X}_0 = \langle (Insert, T_0(1)) \rangle$ $\mathcal{X}_1 = \langle (Insert, T_0(3)) \rangle$

E: $\{ (Insert, T_0) \}$

C: $\neg T_0(3)$ $\mathcal{D}_0 = \{\}$

A: $\langle (Insert, T_1(2)) \rangle$

FIGURE 14. Coupling Modes and Rule Execution.

First, consider when R is stated in E-C and C-A immediate coupling modes (Table 5). In this scenario, $\mathcal{X}_0$ is applied in $\mathcal{D}_0$. The execution of $\mathcal{X}_0$ triggers R. Since R is stated in E-C immediate mode, R's condition is immediately evaluated before $\mathcal{X}_1$ is allowed to occur. Since T_0 is empty, R's condition is satisfied, and since R is stated in C-A immediate mode, R's action immediately inserts the value of 2 into T_1. At this point, the database contains the values 1 in T_0 and 2 in T_1. Formally, the database is in the state $\mathcal{D}_2 = \{T_0(1), T_1(2)\}$ where R is no longer triggered. Thus, $\mathcal{X}_1$ now executes and inserts 3 into T_0. Again, R is triggered. Since R is stated in E-C immediate mode, R's condition is immediately evaluated. However, R's condition is not satisfied since the value 3 exists in T_0. Therefore, the final database state in this scenario is $\mathcal{D}_4 = \{T_0(1), T_0(3), T_1(2)\}$.

State	Values	Operation
$\mathcal{D}_0$	{}	$\mathcal{X}_0$
$\mathcal{D}_1$	$\{T_0(1)\}$	R
$\mathcal{D}_2$	$\{T_0(1), T_1(2)\}$	$\mathcal{X}_1$
$\mathcal{D}_3$	$\{T_0(1), T_0(3), T_1(2)\}$	R
$\mathcal{D}_4$	$\{T_0(1), T_0(3), T_1(2)\}$	

TABLE 5. Scenario 1

Now consider when R is stated in E-C and C-A decoupled coupling modes (Table 6). Again, $\mathcal{X}_0$ is applied in $\mathcal{D}_0$. The execution

State	Values	Operation
$\mathcal{D}_0$	{}	$\mathcal{X}_0$
$\mathcal{D}_1$	$\{T_0(1)\}$	$\mathcal{X}_1$
$\mathcal{D}_2$	$\{T_0(1), T_0(3)\}$	R
$\mathcal{D}_3$	$\{T_0(1), T_0(3)\}$	

TABLE 6. Scenario 2

of $\mathcal{X}_0$ triggers *R*. Since *R* is stated in E-C decoupled mode, *R*'s condition does not have to be immediately evaluated. Therefore, it is possible for $\mathcal{X}_1$ to execute before *R* evaluates its condition. Consider when this is the case. Therefore, $\mathcal{X}_1$ inserts 3 into T_0. At this point, the database contains the values 1 and 3 in T_0. Formally, the database is in the state $\mathcal{D}_2 = \{T_0(1), T_0(3)\}$ where *R* is triggered. Now the database evaluates *R*'s condition. However, its condition is not satisfied since the value 3 exists in T_0. Therefore the final database state in this scenario is $\mathcal{D}_3 = \{T_0(1), T_0(3)\}$, a different final database state than exhibited in the first scenario.

The above scenarios demonstrate how coupling modes dictate execution behavior. Deciding among the possibilities can become quite difficult when developing active database applications which may contain hundreds of rules. Developers must carefully analyze every rule for the most flexible coupling modes that maintain the correct results. Otherwise, rule execution may not scale to the expected level of multi-user activity. This chapter explains how the LMA restrictions can be exploited to simplify these complexities.

4.2 Background

This section presents background material necessary for this study.

4.2.1 LMAs, Datalog, and Confluence

LMAs are closely related to programs stated in the logical database language Datalog [29]. Datalog programs are deductive logic programs with the following properties [90]:

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

The above properties of Datalog significantly overlap with the properties of LMAs.

A number of studies use the above characteristics and slightly differing semantics to map active rules to Datalog [2,29,81,90]. Of particular interest, Comai and Tanca demonstrate one such model in which an active rule set that contains rules with only positive variables and insertions in actions is translated into an equivalent Datalog program [29][2]. They then use the properties of Datalog to prove that the translated program always terminates and is *confluent* (Section 2.2.5.1).

This theory is a foundation on which this study builds its concurrency schemes. Yet, it is not all encompassing. Foremost, an underlying assumption of the previous theories on rule confluence is that rules are executed atomically and in isolation from other database activity. On the other hand, active databases assume a multi-user environment where rules and external events execute in parallel within the semantics of coupling modes. Even so, the studies that have been performed on active databases have concluded with restrictive results [2,90].

Secondly, confluence cannot be guaranteed for ECA rule programs in which rules do not monitor for all events (as is the case for expert system rules).

---

2. They call such rules ECA^+ rules. This paper refers to ECA^+ rules as LMA^+ rules.

However, without loss of generality, this situation is ignored. 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 language semantics.

4.2.2 Previous Work

The previous work in active database language semantics focuses on either 1) language construction (Section 2.2.5, Section 3.3), and/or 2) properties about the termination of active database programs (Section 2.2.5.1). This chapter draws from both domains. In this respect, formal language semantics are used in the construction of this chapter's execution models, while termination properties are exploited in the development of its concurrency schemes.

4.3 Approach

The study begins with a formal specification of the active database languages presented in [7,79,98]. Section 4.4 presents the definitions used in this chapter as well as a unified general-purpose active database language. Section 4.5 expands this language with three increasingly concurrent active database 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 stand alone applications with no other database activ-

ity. 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, its single user environment is impractical. The sequential and atomic properties of the model lead to unacceptable performance with little ability to scale.

The second model presented is the *parallel execution model*. This model expands on the sequential execution model by allowing concurrent rule execution. Although restricting external event behavior reduces the usefulness of this model, the properties proven about the parallel model are used as a stepping stone to prove properties about concurrent LMA 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 [79]. As such, the active database execution model accurately portrays modern active database systems executing within a multi-user environment.

Using these three execution models, this chapter presents a series of proofs that specify the concurrency schemes for LMAs. 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 some path within the sequential execution model (Section 4.6). The analysis is divided into two categories. The first category consists of LMA^+ programs, LMAs that contain only *positive* variables[3]. The second category consists of LMA^- programs, LMAs that contain both positive and *negated* variables[4]. It follows from these definitions that the logical database languages Datalog and DatalogNeg are

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

proper subsets of LMA^+ and LMA^- programs respectively [90]. The proofs exploit the previous results and proof techniques in serializability theory, rule dependency graphs, and confluent rules systems [2,29,60,90].

Serializability theory determines 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 [59]. In [12], Bernstein states a set of conditions that specify when the execution order of interfering operations matter (RAW, WAW, and WAR). These conditions are violated when interfering operations execute in parallel. The results of violating the Bernstein conditions are that the database may move into an incorrect state.

This study exploits serializability theory to describe when the parallel execution of rules interfere with one another. Rule interference is synonymous to the Bernstein conditions where if certain conditions are violated, the order of rule execution matters. This chapter uses Kuo et al.'s rule serializability theory based on bipartite rule dependency graphs (Section 4.7) [56,60].

In Kuo et al., a graph in which a cycle of rules *interfere* with one another is called a *cycle of dependency.* The set of rules in a cycle of dependency form a *mutual exclusion set*. Kuo et al. present two key theorems that describe execution cycles in terms of *cycle serializability*, a parallel execution cycle that is equivalent to some sequence of serially executed rules [60]. 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 of 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 (Section 2.2.5.1) [2,29,81,90]. A rule system is said to be confluent when the quiescent state is unique despite rule ordering. Towards this end, this study presents the program characteristics and concurrency schemes that are sufficient for LMAs to be confluent.

4.3.1 Results

The first result establishes a concurrency scheme for LMA^+ programs (Section 4.8). Theorems 2 and 4 prove that an LMA^+ program is correct when all rules are specified using a single pair of coupling modes, E-C and C-A decoupled as reviewed in Section 4.5.1. In fact, a property of pure Datalog programs is that they are confluent [29]. Theorems 1 and 3 use the similarities of LMAs to Datalog, described in Section 4.2.1, to demonstrate that confluence also holds for active LMA^+ programs. Since decoupled modes maximize concurrency, it is fair to conclude that DBMS's need to only support decoupled coupling modes in order to support the execution of LMA^+ programs.

The next findings concern the more general LMA^- programs (Section 4.9). Concurrency schemes for LMA^- programs are difficult to assign since they do not contain unique quiescent states [29,90]. 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. These complications become apparent within applications where it is possible for an incorrect program execution to contain only cycle serializable execution cycles. Due to such complications, Section 4.9.1 defines *event sequenc-*

ing and *event isolation*. These properties are exploited in the proofs of correctness for LMA$^-$ concurrency schemes.

The LMA$^-$ analysis begins with the concurrency scheme for programs executing the parallel execution model (Section 4.9.2). Kuo et al.'s work is exploited to identify rules that must be stated in E-C and C-A immediate modes. Specifically, Theorem 5 proves that an LMA$^-$ 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.

The study next analyzes LMA$^-$ programs executing with the active database execution model (Section 4.9.3). Three decreasingly restrictive concurrency schemes are presented. 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 may be pictured as all rules reachable by a depth first traversal in the rule dependency graph rooted by the external event.

The first concurrency scheme for LMA$^-$ programs executing with the active database execution model is overly restrictive. Theorem 6 proves that an LMA$^-$ 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 LMA$^-$ programs executing the active database execution model improves concurrency based on transaction characteristics. This study's definition of external events is that they are atomic and committed. Theorem 7 exploits this definition by proving that an LMA$^-$ 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.

The third and most general concurrency scheme for LMA$^-$ 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 not appear to be correctly sequenced. Lemma 4 establishes dependency graph regions where external event interference may occur. Theorem 8 proves that an LMA$^-$ 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 another are stated in E-C and C-A deferred modes or stronger. It is important to note that many LMAs are embedded applications that have a limited number of external events. This last concurrency scheme exploits this property to improve system throughput.

4.4 Definitions

A **database table** is defined 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** [7]. 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*.

An active database **rule base**, $\mathcal{R}$, is defined 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 relational calculus predicate ranging over the extensional database[5]. Variables within the predicate may be either 1) *positive* or 2) *negated*. Positive variables are existentially quantified variables. Negated variables are identical to negation used in Datalog and expert systems languages that use the closed world assumption to test for the absence of values, or in the vernacular, there-does-not exist tests.

5. Relational calculus predicates are assumed to be safe [90].

- 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.)

For a rule $R = (E,C,A)$, the notation E^R, C^R, A^R is sometimes used to denote the rule's constituent parts.

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

Modifications to the database may occur outside of rule execution through an **external event**. An external event, $\mathcal{X}$, is defined as a non-empty 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, external events are assumed to be committed, i.e., external events may not occur in nested subtransactions that can be rolled back. The rolling back of rule execution is beyond the scope of this study.

Active databases change state over time. Towards this end, an **extensional database state**, $\mathcal{D}$, is defined as the state that consists of all the tuples within all of 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 an extensional database state.
- $\mathcal{T} \subseteq \mathcal{R}$ is the set of triggered rules. (The triggering of rules is discussed below.)

An active database is in a **quiescent state**, $(\mathcal{D}, \varnothing)$, when the set of triggered rules is empty. Similarly, an active database executes rules until **quiescence**, the state in which the set of triggered rules is empty[6]. 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. Towards this end, a rule R **monitors** a table T when $\exists(V, T) \in E^R$. Likewise, a table T is **monitored** if $\exists R \in \mathcal{R}$ such that $\exists(V, T) \in E^R$.

Without loss of generality, the following assumption is made:

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

This 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 this study and are henceforth ignored.

4.4.1 Functions

The following is a list of functions used in this chapter.

$C^R(\mathcal{D})$: 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, C^R is said to be *satisfied*.

$C^R(\mathcal{D})$ = *false* otherwise.

---

6. Quiescence is equivalent to fixed-point described in Section 2.1.

$A^R(\mathcal{D})$: Where $\in \mathcal{R}$ 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 $\in \mathcal{E}$.

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

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

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

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

$T \in Neg(C^R)$ = *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 cause 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 $\mathcal{D}'$.

4.4.2 Sequence of States

An active database may move from state $(\mathcal{D}_n,\mathcal{I}_n)$ to state $(\mathcal{D}_{n+1},\mathcal{I}_{n+1})$ 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[7]

i. $C^R(\mathcal{D}_n)$ evaluates to *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. $C^R(\mathcal{D}_n)$ evaluates to *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. $\mathcal{I}_n \cup Triggers(\mathcal{X}) = \mathcal{I}_{n+1}$

7. Due to the properties of LMAs, rules that are un-triggered as described in [97] do not have to be considered.

An **execution graph** is defined 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.

4.4.3 Log Monitoring Application Definitions

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

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

Informally, a table T is an LMA table 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.

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

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

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

Log Monitoring Application (LMA) - 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 an LMA iff:

$$\forall T \in \mathcal{E},\ T \text{ is satisfied by Equation 1.} \qquad \text{(EQ 3)}$$

Equation 3 implies that all rules in an LMA are also LMA rules.

This study distinguishes two categories of LMAs. The first category, **LMA$^+$** programs, are LMAs containing only positive variables in rule conditions. The second category, **LMA$^-$** programs, are LMAs containing both positive and negated variables in rule conditions.

4.5 Active Database Execution

This section presents three slightly different execution models that vary depending on their restrictiveness with respect to concurrency [7,78]. In all three models, rule execution begins with the occurrence of an external event.

The first model, the *Sequential* model, forms the basis of correctness. This simple model, derived from the algorithms presented in [7,8,97], proceeds by locking the database from external events and serially executing rules until quiescence.

Though straightforward, the *Sequential* model forfeits concurrency. Therefore, this section introduces 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 [7,78], allows for concurrent execution of both external events and rules.

Before introducing the execution models, a discussion about the semantics of parallel rule execution and external events is presented.

4.5.1 Atomicity and Parallel Rule Execution

Section 4.4.2 presented the linking of active database states as if rules are executed atomically. However, this is not the case. Operations within an exten-

sional database are guaranteed to be atomic iff the operations execute within a single transaction. In an active database, the set of atomic operations are expanded to include rule conditions, rule actions, and 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 rule evaluation is not atomic, condition and action evaluation may be split into separate parts within a transaction or over multiple transactions. Thus, 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 transaction 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 nested sibling transactions.

3. Conditions in E-C decoupled mode are executed in independent top transactions.

Statements 1 through 3 are identical for rule actions [33].

The definition above results in the 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 without being broken into pieces). Stated formally,

$(\mathcal{D}_k,\mathcal{T}_k) \longrightarrow (\mathcal{D}_{k+1},\mathcal{T}_{k+1})$ represents moving the database from $(\mathcal{D}_n,\mathcal{T}_n)$ to $(\mathcal{D}_{k+1}, \mathcal{T}_{k+1})$ where X is either an external event or rule stated in E-C and C-A immediate modes.

2. Parallel transition (also called a *parallel execution cycle*). Stated formally, $(\mathcal{D}_k,\mathcal{T}_k) \xrightarrow{X} (\mathcal{D}_{k+1},\mathcal{T}_{k+1})$ represents moving the database from $(\mathcal{D}_n,\mathcal{T}_n)$ to $(\mathcal{D}_{k+1}, \mathcal{T}_{k+1})$ where $X = \{\mathcal{R}_0, \ldots, \mathcal{R}_x \cup \mathcal{X}_0, \ldots, \mathcal{X}_y\}$, $\mathcal{R}_i \in \mathcal{R}$, $\mathcal{X}_j \in \mathcal{X}$. The algorithm for a parallel transition with the above set X is as follows:

   while($X \neq \{\ \}$)

      do_in_parallel

      choose R from X, and remove it from X.

      R links $(\mathcal{D}_k,\mathcal{T}_k)$ to $(\mathcal{D}_k',\mathcal{T}_k')$ 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 this chapter is to clarify the meaning of $(\mathcal{D}_{k+1},\mathcal{T}_{k+1})$.

4.5.2 Execution Models

The following are the definitions of our three execution models.

4.5.2.1 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) $(\mathcal{D}_i, \varnothing) \longrightarrow (\mathcal{D}_{i+1}, \mathcal{I}_{i+1})$, $i := i + 1$

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

Begin loop

2) Select $\in \mathcal{I}_i$

3) $(\mathcal{D}_i, \mathcal{I}_i) \longrightarrow (\mathcal{D}_{i+1}, \mathcal{I}_{i+1})$, $i := i + 1$

End loop

4) return $\mathcal{D}_i$

4.5.2.2 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) $(\mathcal{D}_i, \varnothing) \longrightarrow (\mathcal{D}_{i+1}, \mathcal{I}_{i+1})$, $i := i + 1$

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

Begin loop

2) Select $R = \mathcal{I}_i$

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

End loop

4) return $\mathcal{D}_i$

Step 2 of the *Parallel* model has been the subject of much research [21,87,97]. Many of the first active database languages modeled the *Sequential* algorithm where one rule is selected from T_k on each cycle [97]. *Parallel* improves system throughput by allowing rules to execute concurrently. The *Parallel* model, similar to the REACH rule system [21], selects **all** rules in T_k for concurrent execution on each cycle. This choice may seem overly aggressive since if all the rules were really executed in parallel, incorrect behavior could result. However, this study shows that 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.

4.5.2.3 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) $(\mathcal{D}_i, \varnothing) \longrightarrow (\mathcal{D}_{i+1}, \mathcal{T}_{i+1})$, $i := i + 1$

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

Begin loop

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

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

End loop

4) 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.

To simplify notation, the remainder of this study refers to an active database program Y that executes using the execution model X beginning in state $(\mathcal{D}_i, \varnothing)$ executing a sequence of external events $\mathcal{X}$ and terminating in state $(\mathcal{D}_j, \varnothing)$ as:

$$X_Y(\mathcal{D}_i, \mathcal{X}) \Rightarrow \mathcal{D}_j$$

4.6 Correct Active Database Execution

Proofs of concurrency schemes must demonstrate that the scheme enforces correct active database behavior. Towards this end, this section defines correct active database execution.

An active database state $\mathcal{D}_j$ is defined 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. Note that the initial database state, $(\mathcal{D}_0,\varnothing)$, is a correct active database state, it is the state in which no data manipulation commands have occurred.

The above definition of a sequential database state combined with the simplicity of the *Sequential* execution model gives rise to the following corollary that is used to form this study's definition of program correctness:

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 Corollary 1, program correctness is defined per its execution using 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. Therefore, $Correct_Y(\mathcal{D}_n,\mathcal{X})$ is defined as the **set** of quiescent states reachable by all possible execution paths of $Sequential_Y(\mathcal{D}_n, \mathcal{X})$. The states in $Correct_Y(\mathcal{D}_n,\mathcal{X})$ are referred to as correct quiescent states. Stated formally, for an active database program *Y* that begins execution in state $\mathcal{D}_n$ and processes the sequence of external events

$\mathcal{X} = \{\mathcal{X}_0, \ldots, \mathcal{X}_{n-1}\}$,

$$Correct_Y(\mathcal{D}_n,\mathcal{X}) = \{\text{states } S \mid \exists \text{ an execution } Sequential_Y(\mathcal{D}_n, \mathcal{X}) \Rightarrow S\} \quad \text{(EQ 4)}$$

Now we are ready for the formal definition for a correct active database program.

Correct Active Database Program - Consider an active database program *Y* that begins execution 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 $Correct_Y(\mathcal{D}_i, \mathcal{X})$. A program *Y* is *correct* under an 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 $\mathcal{D}_{n+k} \in Correct_Y(\mathcal{D}_n, \mathcal{X})$.

4.7 Serializability of Rules

This study heavily exploits the standard definitions and methodology surrounding the serializability theorem [59]. However, its manifestation in active databases and our presentation requires some review and adaptation.

Serializability theory determines 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 [59]. In [12], Bernstein states a set of conditions that specify when the execution order of interfering operations matter (RAW, WAW, and WAR). These conditions are violated when interfering operations execute in parallel. The results of violating the Bernstein conditions is that the database may move to an incorrect state.

The evaluation of individual rules may be explored with respect to the Bernstein conditions. In this case, the parallel execution of interfering rules may produce an incorrect state. Formally, for an LMA, a rule R_0 *interferes* with a rule R_1 iff

$$\exists(Insert, T) \in A^{R_0} \mid \exists(Insert, T) \in Neg(C^{R_1}) \qquad \text{(EQ 5)}$$

External events may also interfere with rules. For an LMA, an external event $\mathcal{X}$ *interferes* with a rule R iff

$$\exists(Insert, T) \in \mathcal{X} \mid \exists(Insert, T) \in Neg(C^{R}) \qquad \text{(EQ 6)}$$

In this paper, a simplified version of the bipartite dependency graphs developed in [60] is used to statically determine rule interferences[8]. A dependency

8. The simplifications result from the properties of LMAs.

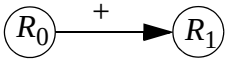
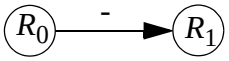
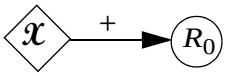
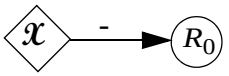
Edge	Definition
1). 	R_1 monitors R_0; R_0 and R_1 can **not** interfere with one another.
2). 	R_1 monitors R_0; R_0 and R_1 interfere with one another.
3). 	R_0 monitors $\mathcal{X}$; $\mathcal{X}$ and R_0 can **not** interfere with one another.
4). 	R_0 monitors $\mathcal{X}$; $\mathcal{X}$ and R_0 interfere with one another.

FIGURE 15. Bipartite Graph Constructs

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 15, represent the data dependency between rules and external events.

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 on 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 [60] used in this study are:

Cycle Serializability Theorem. The parallel execution of all the rules within a mutual exclusion set may not be *cycle serializable*. Proof given in [60].

Serializability Theorem. A parallel execution cycle that does not contain all of the rules within a mutual exclusion set is guaranteed to be *cycle serializable*. Proof given in [60].

These theorems are used 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.

4.8 Concurrency Schemes for LMA$^+$ Programs

This section presents the concurrency schemes for LMA$^+$ programs. It begins with a discussion on programs executing the *Parallel* execution model and concludes 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. The resulting concurrency schemes demonstrate that LMA$^+$ programs with all rules stated in E-C and C-A decoupled modes are correct and confluent.

4.8.1 Parallel Execution Model

Lemma 1. Given an LMA$^+$ 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 suffices to prove the more general claim that all parallel execution cycles in all execution paths of an LMA$^+$ program 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.

Steps 0-2 and Step 4 in the parallel execution model are cycle serializable by definition (page 67). Now it is necessary to prove that Step 3 in the parallel execution model evaluating an LMA$^+$ program is cycle serializable. Construct 1 in Figure 15 is the only edge notation connecting rules in LMA$^+$ programs. 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. □

4.8.1.1 Confluence

Theorem 1. The execution of an LMA$^+$ 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 LMA$^+$ program executing the *Parallel* execution model (with the stated coupling modes) are cycle serializable. Therefore, LMA$^+$ 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. Comai and Tanca prove that active rules that contain only positive variables and insertions in rule conditions are confluent [29]. In its most restrictive case, their execution model reduces to the *Sequential* execution model. Therefore, in the sequential case, the programs that Comai and Tanca prove confluent are equivalent to sequential LMA$^+$ programs. Thus, by transitivity, sequential LMA$^+$ programs are confluent, and LMA$^+$ programs using the *Parallel* execution model are also confluent. □

4.8.1.2 Program Correctness

Theorem 2. The execution of an LMA$^+$ 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 LMA$^+$ program *Y* in which all rules are stated in E-C and C-A decoupled modes, $Parallel_Y(\mathcal{D}_n, \mathcal{X})$ is correct, by definition, iff

i. all parallel execution cycles in all execution paths are cycle serializable, and

ii. 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 *i*) 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 *ii*). Theorem 1 proves that LMA^+ programs using the *Parallel* execution model are confluent. Therefore, $Parallel_Y(\mathcal{D}_n, \mathcal{X}) = Correct_Y(\mathcal{D}_n, \mathcal{X})$ since there is only one quiescent state in a confluent program.

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

4.8.2 Active Database Execution Model

Lemma 2. Given an LMA^+ 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. The proof of Lemma 2 is almost identical to that of Lemma 1 since external events only add constructs 3 and 5 in Figure 15 to the dependency graphs. Specifically, it suffices to prove the more general claim that all parallel execution cycles in all execution paths of an LMA^+ program using the *ActiveDatabase* execution model are cycle serializable (regardless of coupling modes).

Steps 0-2 in the *ActiveDatabase* execution model are cycle serializable by definition (page 68). Now it is necessary to prove that Step 3 in the *ActiveDatabase* execution model evaluating an LMA^+ program is cycle serializable. Con-

structs 1, 3, and 5 in Figure 15 are the only edges connecting rules in LMA$^+$ programs 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. □

4.8.2.1 Confluence

Theorem 3. The execution of an LMA$^+$ 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 LMA$^+$ program executing the *ActiveDatabase* execution model (with the stated coupling modes) are cycle serializable. Therefore, LMA$^+$ 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. Comai and Tanca prove that active rules that contain only positive variables and insertions in rule conditions are confluent [29]. In its most restrictive case, their execution model reduces to the *Sequential* execution model. Therefore, in the sequential case, the programs that Comai and Tanca prove confluent are equivalent to sequential LMA$^+$ programs. Thus, by transitivity, sequential LMA$^+$ programs

are confluent, and LMA$^+$ programs using the *ActiveDatabase* execution model are also confluent. □

4.8.2.2 Program Correctness

Theorem 4. The execution of an LMA$^+$ 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 LMA$^+$ program *Y* in which all rules are stated in E-C and C-A decoupled modes, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X})$ is correct, by definition, iff

i. all parallel execution cycles in all execution paths are cycle serializable, and

ii. all executions of $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ 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 2 satisfies *i*) 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 *ii*). Theorem 2 proves that LMA$^+$ programs using the *ActiveDatabase* execution model are confluent. Therefore, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) = Correct_Y(\mathcal{D}_n, \mathcal{X})$ since there is only one quiescent state in a confluent program.

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

4.9 Concurrency Schemes for LMA⁻ Programs

This section considers LMA⁻ programs. In general, LMA⁻ programs are not confluent since rules and external events may interfere with one another [90,98].

In this study, an active database program is defined to be 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, proofs of correctness must present the sufficient conditions to maintain the ordering of external events[9].

This section begins with a discussion of external event sequencing and isolation and their impact to proofs on correct active database programs.

4.9.1 External Event Sequencing and Isolation

External events appear as if they are processed in sequence if the events are truly sequenced or they are isolated from one another such that their processing order does not matter. In this respect, an active database program that executes a sequence of external events $\mathcal{X}$ will be correct if all execution cycles are cycle serializable, and all external events within $\mathcal{X}$ are sequenced or isolated. More formally, an external event $\mathcal{X}_0$ is processed in **sequence** before an external event $\mathcal{X}_1$ iff

9. This sequencing has not been necessary in previous sections. For example, Section 4.8 presents LMA⁺ programs that are confluent. Confluence means sequence is irrelevant.

- rule evaluation spawned from $\mathcal{X}_0$ quiesces before (in time) the rule evaluation spawned from $\mathcal{X}_1$.

To define external event isolation, the definitions of the closure of rules must be defined. For $R \subseteq \mathcal{R}$, the set of rules within the *Closure*(*R*) is defined by the following algorithm:

Algorithm: *Closure*

Input: $R \subseteq \mathcal{R}$

Ouput: $S \subseteq \mathcal{R}$

$S \leftarrow \{R\}$

Repeat until *S* is unchanged:

$S \leftarrow S \cup X \in \mathcal{R} \mid X \in Triggers(A^Y)$ 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 *R*. For convenience, the $Closure(\mathcal{X})$ for a sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \ldots, \mathcal{X}_{m-1}\}$ is equivalent to the $Closure(Triggers(\mathcal{X}_0) \cup \ldots \cup Triggers(\mathcal{X}_{m-1}))$.

Now we can define **external event isolation**. External events $\mathcal{X}_i$ and $\mathcal{X}_j$, $i \neq j$, are *isolated* from one another when either $Closure(\mathcal{X}_i) \cap Closure(\mathcal{X}_j) = \varnothing$, or there does not exist a negative edge in the dependency graph to or from any rule within $Closure(\mathcal{X}_i) \cup Closure(\mathcal{X}_j)$.

These definitions lead to the following corollary.

Corollary 2. For an LMA$^-$ program Y and sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \ldots, \mathcal{X}_{m-1}\}$, if all the external events within $\mathcal{X}$ are *isolated* from one another, then any execution serializable execution of $ActiveDatabase_Y(\mathcal{D}, \mathcal{X}) \in Correct_Y(\mathcal{D}, \mathcal{X})$ for any initial state $\mathcal{D}$.

Proof. This proof is divided into two cases based on the classification of the rules $A = Closure(\mathcal{X})$. In the first case, there does *not* exist a negated variable in any of the rules in *A.* In this case, the subregions of the dependency graph defined by *A* and $\mathcal{X}$ form LMA$^+$ subregions. Theorem 3 tells us that execution serializable LMA$^+$ programs executing the *ActiveDatabase* execution model are confluent. Therefore, $ActiveDatabase_Y(\mathcal{D}, \mathcal{X}) = Correct_Y(\mathcal{D}, \mathcal{X})$.

In the second case, there *does* exist a negated variable in the rules in *A.* This case can be further divided into two subcases. First, consider the set of external events $\mathcal{X}' \subseteq \mathcal{X}$ such that $\exists i,j \in [0, ..., m\text{-}1]$, $i \neq j$, and $Closure(\mathcal{X}_i) \cap Closure(\mathcal{X}_j) \neq \varnothing$. Since *Y* is an LMA$^-$ program and the external events in $\mathcal{X}$ are isolated from one another, there does not exist a negative edge in the subregions of the dependency graph defined by $B = Closure(\mathcal{X}')$ and the external events $\mathcal{X}'$. Note, $B' \subseteq A$. If *B* contains negated variables, the satisfaction of the negated variables in *B* will *not* be modified by the execution of $ActiveDatabase_Y(\mathcal{D}, \mathcal{X})$ since non of the rules interfere with one another. The set of rules *B* can be divided into the two sets, 1) the rules $I \subseteq B$ that contain negated variables that are invalid due to the table states in $\mathcal{D}$, and 2) the remaining rules $B' \subseteq B - I$. Since *Y* is an LMA$^-$ and none of the rules in *I* interfere with one another, the negated variables within *I* will never again become satisfied (no data is deleted). Thus, *I* can be removed from *B* without modifying the execution of *Y.*

Now consider the remaining rules *B*'. The negated variables in the rules in *B*' have not been invalidated due to the table states in $\mathcal{D}$, and they will not be invalidated by the execution of $ActiveDatabase_Y(\mathcal{D}, \mathcal{X})$ since none of the rules interfere with one another. Thus, the negated variables within the rules in *B*' can be removed during the execution of $ActiveDatabase_Y(\mathcal{D}, \mathcal{X})$ since they also will not effect execution behavior; call this modified set of rules *B*". Note, by construction, $ActiveDatabase_Y(\mathcal{D}, \mathcal{X}')$ produces the same set of quiescent states whether it executes over the rules *B* or *B*". Since *B*" contains only positive variables, it is equivalent to an LMA^+ program. Theorem 3 tells us that execution serializable LMA^+ programs executing the *ActiveDatabase* execution model are confluent. Therefore, $ActiveDatabase_Y(\mathcal{D}, \mathcal{X}') = Correct_Y(\mathcal{D}, \mathcal{X}')$.

Now consider the second subcase. This subcase is defined by the set of external events in $\mathcal{X}'' = \mathcal{X} - \mathcal{X}'$. Since this subcase is the contrapositive of the first subcase, it can be concluded that $\forall i, j \in [0, \ldots, m-1]$, if $i \neq j$ and $\mathcal{X}_i \in \mathcal{X}''$, then $Closure(\mathcal{X}_i) \cap Closure(\mathcal{X}_j) = \varnothing$. In this subcase, since there is no interaction between the execution of the external events within $\mathcal{X}''$ and any other external event within $\mathcal{X}$, any interleaved execution serializable execution of the external events within $\mathcal{X}''$ will produce the same set of quiescent states. Therefore, $ActiveDatabase_Y(\mathcal{D}, \mathcal{X}'') \in Correct_Y(\mathcal{D}, \mathcal{X}'')$. Thus,

$ActiveDatabase_Y(\mathcal{D}, \mathcal{X}) \in Correct_Y(\mathcal{D}, \mathcal{X})$ since

$ActiveDatabase_Y(\mathcal{D}, \mathcal{X}') \in Correct_Y(\mathcal{D}, \mathcal{X}')$,

$ActiveDatabase_Y(\mathcal{D}, \mathcal{X}'') \in Correct_Y(\mathcal{D}, \mathcal{X}'')$, $\mathcal{X} = \mathcal{X}' \cup \mathcal{X}''$, and there is no interaction between $ActiveDatabase_Y(\mathcal{D}, \mathcal{X}')$ and $ActiveDatabase_Y(\mathcal{D}, \mathcal{X}'')$.

Both cases have been proven, thus the corollary is proved. □

External event isolation and sequencing are interesting since the *Sequential* execution model processes an external event $\mathcal{X}$ by locking the active database and sequentially evaluating rules within the $Closure(\mathcal{X})$ until quiescence. Therefore, it is sufficient for a proof of program correctness to prove that all execution cycles are cycle serializable, and all external events are either sequenced or execute in isolation from one another.

Example 2. This example demonstrates the necessity for external event sequencing. Consider Example 1 presented in Section 4.1.2. Example 1 introduces two external events $\mathcal{X}_0$ and $\mathcal{X}_1$ and a single rule *R* (Figure 14). Figure 16 illustrates the resulting dependency graph. Notice that $\mathcal{X}_0$ and $\mathcal{X}_1$ are not isolated from one another. Therefore, Section 4.1.2 demonstrates that when *R* is stated in E-C and C-A immediate modes (Scenario 1), the quiescent state may be different than the quiescent state produced when *R* is stated in E-C and C-A decoupled modes (Scenario 2). In the E-C and C-A decoupled modes scenario, the presented execution is not consistent with the experienced sequence of the external events and any execution path in the *Sequential* execution model. Therefore, it is incorrect, and $\mathcal{X}_0$ and $\mathcal{X}_1$ need to be sequenced.

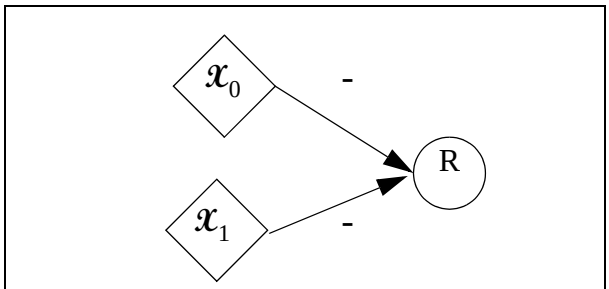


FIGURE 16. Dependency Graph of Figure 14

We are now ready to present our concurrency schemes for LMA$^-$ programs.

4.9.2 Parallel Execution Model

4.9.2.1 Cycle Serializability

Lemma 3. Given an LMA$^-$ 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.

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. Steps 0-2 and Step 4 are cycle serializable by definition (page 67). Now it is necessary to prove that Step 3 in the *Parallel* execution model evaluating an LMA$^-$ 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 of these rules to E-C and C-A immediate modes. According to the definitions in Section 4.5.1, 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 3. Given an LMA$^-$ 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. Lemma 3 proves that the conditions of the corollary are sufficient to guarantee cycle serializability in the *Parallel* execution model. For the purpose of contradiction, suppose that a *weaker* concurrency scheme exists that also statically guarantees cycle serializability within the *Parallel* execution model, i.e., consider a concurrency scheme that contains mutual exclusion sets without any rules stated in E-C and C-A immediate modes that still guarantees cycle serializability. In this weaker scheme, consider one of these mutual exclusion sets that does not contain any rules stated in E-C and C-A immediate modes. In such a set, a possible interleaving of operations include evaluating all the rule conditions in the mutual exclusion set before executing any actions. This interleaving is possible because in this particular set of rules:

- Rules stated using an E-C immediate mode must have a C-A deferred mode or weaker.

- Rules stated using an C-A immediate mode must have a E-C deferred mode or weaker.

- All decoupled and deferred modes may be evaluated in parallel.

The above conditions imply that the rules in the identified mutual exclusion set could be evaluated using the following algorithm:

1. Evaluate all the rule conditions in the exclusion set that have an E-C immediate coupling mode.

2. Evaluate all the rule conditions in the exclusion set that are stated in either E-C deferred or decoupled mode whose C-A coupling mode is not immediate.

3. Evaluate in parallel the remaining conditions.

4. Perform the actions of all satisfied rules according to their coupling modes.

In the above algorithm, it is possible that no action will be executed until all the rule conditions have been evaluated. This described interleaving is equivalent to executing all the rules in the mutual exclusion set in parallel since all rules will evaluate their respective conditions in the same database state and execute their LMA rule 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, and therefore, the corollary has been proven. □

Example 3. To illustrate Corollary 3, consider the example presented in Figure 17. 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 absence of other external events, a legal interleaving of operations is to apply $\mathcal{X}_0$ in state $\mathcal{D}_0$, evaluate C^{R_1} in $\mathcal{D}_1$, evaluate C^{R_0}

rule R_0	rule R_1	$\mathcal{D}_0 = \{\ \}$
E: { (*Insert*,T_0) }	E: { (*Insert*,T_1) }	$\mathcal{X}_0 = \langle$ (*Insert*,T_0(3)),
C: $\neg T_0(1)$	C: $\neg T_1(1)$	(*Insert*,T_1(3)) $\rangle$
A: $\langle$ (*Insert*,T_1(1)) $\rangle$	A: $\langle$ (*Insert*,T_0(1)) $\rangle$	

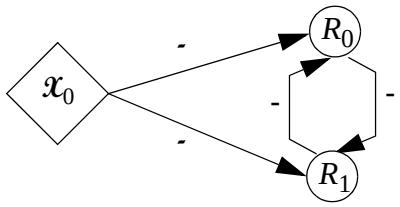


FIGURE 17. An example cycle of dependency.

in $\mathcal{D}_2$, and execute A^{R_1} and A^{R_0} in $\mathcal{D}_3$ and $\mathcal{D}_4$ respectively. 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.

4.9.2.2 Program Correctness

Theorem 5. The execution of an LMA- 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 to the proof of Theorem 4. Specifically, given any LMA- program *Y* in which all rules are stated in the coupling modes described by the theorem condition, $Parallel_Y(\mathcal{D}_n, \mathcal{X})$ is correct, by definition, iff

i. all parallel execution cycles in all execution paths are cycle serializable, and

ii. 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 *i*) 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 *ii*). Consider the *Parallel* execution model. By definition, the *Parallel* execution model processes each external event sequentially until quiescence. Since *i*) proved that all parallel execution cycles are cycle serializable, it can be concluded that for all executions of *Y*, $Parallel_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$.[10]

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

4.9.3 ActiveDatabase Execution Model

Lemma 4. Given an LMA$^-$ 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 LMA$^-$ programs using the *ActiveDatabase* execution model versus the *Parallel* execution model, with respect to cycle

10. In fact, the sufficient conditions for confluent LMA$^-$ programs are shown in [29,81]. These studies demonstrate the transformations upon *stratified* active database rules, programs that contain no cycles of interference, to obtain confluence.

serializability, is that parallel execution cycles may contain external events. Yet, external events do not introduce non-serializable 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[11]. 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 theorem condition are cycle serializable. □

4.9.3.1 Program Correctness

Three concurrency schemes are presented for the correctness of programs executing the *ActiveDatabase* execution model. Each successive scheme allows for more concurrency.

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

Theorem 6. The execution of an LMA$^-$ 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 $\mathcal{X}_i$ in which the *Closure*($\mathcal{X}_i$) contains a rule connected with a negative edge in the dependency graph, all of the rules in the *Closure*($\mathcal{X}_i$) are stated in E-C and C-A immediate modes.

11. 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.

Proof. The proof of Theorem 6 is similar to the proof of Theorem 5. Specifically, given any LMA$^-$ program *Y* in which all rules are stated in the coupling modes described by the theorem conditions, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X})$ is correct, by definition, iff

i. all parallel execution cycles in all execution paths are cycle serializable, and

ii. all executions of $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ 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 *i*) 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 *ii*). The theorem conditions specify that for every external event $\mathcal{X}_i$ in which the *Closure*($\mathcal{X}_i$) contains a rule connected with a negative edge in the dependency graph, all of the rules in the *Closure*($\mathcal{X}_i$) are stated in E-C and C-A immediate modes. Label this set of external events $\mathcal{X}'$. The external events in $\mathcal{X}'$ are necessarily sequenced since for any $\mathcal{X}_i \in \mathcal{X}'$, all of the rules in *Closure*($\mathcal{X}_i$) are executed atomically within a single transaction. Since *i*) proved that all parallel execution cycles are cycle serializable, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}') \in Correct_Y(\mathcal{D}_n, \mathcal{X}')$ by definition.

Now consider the remaining set of external events $\mathcal{X}''$. These external events do not trigger any interfering rules. Thus, by definition, all of the external events in $\mathcal{X}''$ are isolated from one another. Corollary 2 says that since

$ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'')$ is an execution serializable execution of a set of isolated external events, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'') \in Correct_Y(\mathcal{D}_n, \mathcal{X}'')$.

Lastly, the theorem's conditions imply that $Closure(\mathcal{X}') \cap Closure(\mathcal{X}'') = A$ does not contain any rules connected with a negative edge. Otherwise, the offending rules and their closures would be in $\mathcal{X}$' and not $\mathcal{X}$". Thus, using the same logic presented in Corollary 2, if *A* is non-empty, *A* can be reduced to an LMA^+ set of rules. Since LMA^+ regions contain no interfering rules and Theorem 3 proves that these regions are confluent, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}')$ and $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'')$ can be interleaved in any order without affecting the resulting database states. Therefore, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for all possible execution paths.

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

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

Theorem 7. The execution of an LMA^- 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 $\mathcal{X}_i$ in which the *Closure*($\mathcal{X}_i$) contains a rule connected with a negative edge in the dependency graph, all of the rules in the *Closure*($\mathcal{X}_i$) are stated in E-C and C-A deferred modes or stronger.

Proof. This proof is similar to the proof of Theorem 6. Specifically, given any LMA$^-$ program *Y* in which all rules are stated in the coupling modes described by the theorem conditions, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X})$ is correct, by definition, iff

i. all parallel execution cycles in all execution paths are cycle serializable, and

ii. all executions of $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for any initial state $\mathcal{D}_n$ and any sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \mathcal{X}_1, \ldots \mathcal{X}_{n-1}\}$.

i) is proved identically as in Theorem 6. Therefore, it is only left to prove that the loosened conditions of Theorem 7 are still sufficient for *ii*).

The theorem conditions specify that for every external event $\mathcal{X}_i$ in which the *Closure*($\mathcal{X}_i$) contains a rule connected with a negative edge in the dependency graph, all of the rules in the *Closure*($\mathcal{X}_i$) are stated in E-C and C-A deferred modes. Label this set of external events $\mathcal{X}'$. The external events in $\mathcal{X}'$ are necessarily sequenced since for any $\mathcal{X}_i \in \mathcal{X}'$, all of the rules in *Closure*($\mathcal{X}_i$) are executed to quiescence before the end of a transaction, and our definition of an external event says that no other external event is permitted to execute until the transaction has been completely committed. Since *i*) proved that all parallel execution cycles are cycle serializable, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}') \in Correct_Y(\mathcal{D}_n, \mathcal{X}')$ by definition.

Now consider the remaining set of external events $\mathcal{X}$". These external events do not trigger any interfering rules. Thus, by definition, all of the external events in $\mathcal{X}$" are isolated from one another. Corollary 2 says that since

$ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'')$ is an execution serializable execution of a set of isolated external events, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'') \in Correct_Y(\mathcal{D}_n, \mathcal{X}'')$.

Lastly, the theorem's conditions imply that $Closure(\mathcal{X}') \cap Closure(\mathcal{X}'') = A$ does not contain any rules connected with a negative edge. Otherwise, the offending rules and their closures would be in $\mathcal{X}$' and not $\mathcal{X}$". Thus, using the same logic presented in Corollary 2, if *A* is non-empty, *A* can be reduced to an LMA^+ set of rules. Since LMA^+ regions contain no interfering rules and Theorem 3 proves that these regions are confluent, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}')$ and $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'')$ can be interleaved in any order without affecting the resulting database states. Therefore, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ 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 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. Borrowing from the definition of event isolation, a close examination of rule dependency graphs reveals that external events need to be sequenced only when rules within their closures interfere with one another. The following is a formal definition of this occurrence.

External Event Interference - Two external events, $\mathcal{X}_i$ and $\mathcal{X}_j$, *interfere* with one 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. In other words, the definition of external event interference is very nearly the contrapositive of the definition of external event isolation.

Corollary 4 follows from this definition.

Corollary 4. For an LMA$^-$ program *Y*, any initial state $\mathcal{D}_n$, and the sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \mathcal{X}_1, \ldots \mathcal{X}_{m-1}\}$ that can *not* interfere with one another, if all cycles are cycle serializable, then $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$.

Proof. Consider the set of external events $\mathcal{X}' \subseteq \mathcal{X}$ such that $\forall i \in [0, \ldots, m-1]$, if $\mathcal{X}_i \in \mathcal{X}'$, then $\mathcal{X}_i$ is isolated from all other external events within $\mathcal{X}$. Corollary 2 tells us that $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}') \in Correct_Y(\mathcal{D}_n, \mathcal{X}')$.

Now consider the remaining set of external events $\mathcal{X}'' = \mathcal{X} - \mathcal{X}'$. By construction, $\forall i \in [0, \ldots, m-1]$, if $\mathcal{X}_i \in \mathcal{X}''$, then $\mathcal{X}_i$ does not interfere with any external event in $\mathcal{X}$, yet $\mathcal{X}_i$ is not isolated from all the external events in $\mathcal{X}$. From the definitions of external event isolation and interference, it can be concluded that some of the rules in $A = Closure(\mathcal{X}_i) - Closure(\mathcal{X} - \mathcal{X}_i)$ may be connected with a negative edge. This is because the only difference between an isolated external event and a non-interfering external event is that a non-interfering external event may contain a negative edge in its closure, even though its closure intersects with other external event closures. If the external event closures do not contain negative edges, then they are isolated from one another. Set *A* is defined by these differences. Thus, the negative edges in *A* can only interfere with execution spawned by $\mathcal{X}_i$. Otherwise, $\mathcal{X}_i$ would interfere with another external event in $\mathcal{X}$. This

implies that for any execution serializable execution, any interleaving of $\mathcal{X}_i$ with any other external event within $\mathcal{X}$ will not modify the satisfiability of the negated variables in *A*. Now consider the rest of the rules in $Closure(\mathcal{X}_i)$, i.e., *B* = $Closure(\mathcal{X}_i)$ - *A*. The rules in *B* do not contain negative edges, otherwise $\mathcal{X}_i$ would interfere with another external event in $\mathcal{X}$. All of the rules in *B* can be reduced to contain only positive variables using the logic presented in Corollary 2. Thus, by Theorem 3, these subregions of the dependency graph are confluent. The properties of *A* and *B* imply that $\mathcal{X}_i$ can be interleaved in any order with the external events in $\mathcal{X}$ without modifying the resulting set of database states. By induction, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'') \in Correct_Y(\mathcal{D}_n, \mathcal{X}'')$.

Lastly, the corollary's conditions imply that $Closure(\mathcal{X}') \cap Closure(\mathcal{X}'') = C$ does not contain any rules connected with a negative edge. Otherwise, the external events would interfere with one another. Thus, using the same logic presented in Corollary 2, if *C* is non-empty, *C* can be reduced to an LMA^+ set of rules. Since LMA^+ regions contain no interfering rules and Theorem 3 proves that these regions are confluent, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}')$ and $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'')$ can be interleaved in any order without affecting the resulting database states. Therefore, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for all possible execution paths.

□

The loosest concurrency scheme for LMA^- programs using the *ActiveDatabase* execution model can now be presented.

Theorem 8. The execution of an LMA$^-$ 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 $\mathcal{X}_i$ that *interferes* with another external event, all the rules in the *Closure*($\mathcal{X}_i$) are stated in E-C and C-A deferred modes or stronger.

Proof. This proof is similar to the proof of Theorem 7. Specifically, given any LMA$^-$ program *Y* in which all rules are stated in the coupling modes described by the theorem conditions, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X})$ is correct, by definition, iff

i. all parallel execution cycles in all execution paths are cycle serializable, and

ii. all executions of $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for any initial state $\mathcal{D}_n$ and any sequence of external events $\mathcal{X} = \{\mathcal{X}_0, \mathcal{X}_1, \ldots \mathcal{X}_{n-1}\}$.

i) is proved identically as in Theorem 6. Therefore, it is only left to prove that the loosened conditions of Theorem 8 are still sufficient for *ii*).

The theorem conditions specify that for every external event $\mathcal{X}_i$ that *interferes* with another external event, all of the rules in the *Closure*($\mathcal{X}_i$) are stated in E-C and C-A deferred modes or stronger. Label this set of external events $\mathcal{X}'$. The external events in $\mathcal{X}'$ are necessarily sequenced since for any $\mathcal{X}_i \in \mathcal{X}'$, all of the rules in *Closure*($\mathcal{X}_i$) are executed to quiescence before the end of a transaction, and

our definition of an external event says that no other external event is permitted execution until the transaction has been completely committed. Since *i*) proved that all parallel execution cycles are cycle serializable, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}') \in Correct_Y(\mathcal{D}_n, \mathcal{X}')$ by definition.

Now consider the remaining set of external events $\mathcal{X}$". All the external events in $\mathcal{X}$" can not interfere with one another. Corollary 4 tells says that $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'') \in Correct_Y(\mathcal{D}_n, \mathcal{X}'')$.

Lastly, the theorem's conditions imply that $Closure(\mathcal{X}') \cap Closure(\mathcal{X}'') = A$ does not contain any rules connected with a negative edge. Otherwise, the offending rules and their closures would be in $\mathcal{X}$' and not $\mathcal{X}$". Thus, using the same logic presented in Corollary 2, if *A* is non-empty, *A* can be reduced to an LMA^+ set of rules. Since LMA^+ regions contain no interfering rules and Theorem 3 proves that these regions are confluent, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}')$ and $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}'')$ can be interleaved in any order without affecting the resulting database states. Therefore, $ActiveDatabase_Y(\mathcal{D}_n, \mathcal{X}) \in Correct_Y(\mathcal{D}_n, \mathcal{X})$ for all possible execution paths.

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

4.10 VenusDB Integration

So far, this chapter has presented concurrency schemes for LMAs executing in a general active database environment. This section applies these techniques to LMAs executing within VenusDB. An algorithm for assigning concurrency schemes is presented. This algorithm can be directly implemented

within the VenusDB compiler resulting in a system that insulates LMA application programmers from the details of concurrency control.

4.10.1 Background

Recall, active rules are evaluated using the nested transaction model as proposed by the HiPAC project (Section 4.5.1) [33]. This behavior can be summarized as:

- Rule constituents in immediate coupling mode spawn sequential sibling nested transactions.
- Rule constituents in deferred coupling mode spawn parallel sibling nested transactions right before a transaction commit.
- Rule constituents in decoupled coupling mode spawn parallel top level transactions.

VenusDB is an active database language that also operates within the nested transaction model (Chapter 3). As such, VenusDB rule conditions and actions execute as an atomic state transition in a state space. However, actions of VenusDB rules may include one or more modules. Such rules that list one or more modules in its action are called *guard rules*. Thus, restrictions must be placed on coupling mode assignments of VenusDB guard rules. These restrictions include the following two rules:

- C-A deferred restriction - Decoupled mode is not allowed for rule constituents in child modules that can be called from a guard rule stated in C-A deferred mode. This ensures that the child modules will execute in the same transaction as the C-A deferred rule.

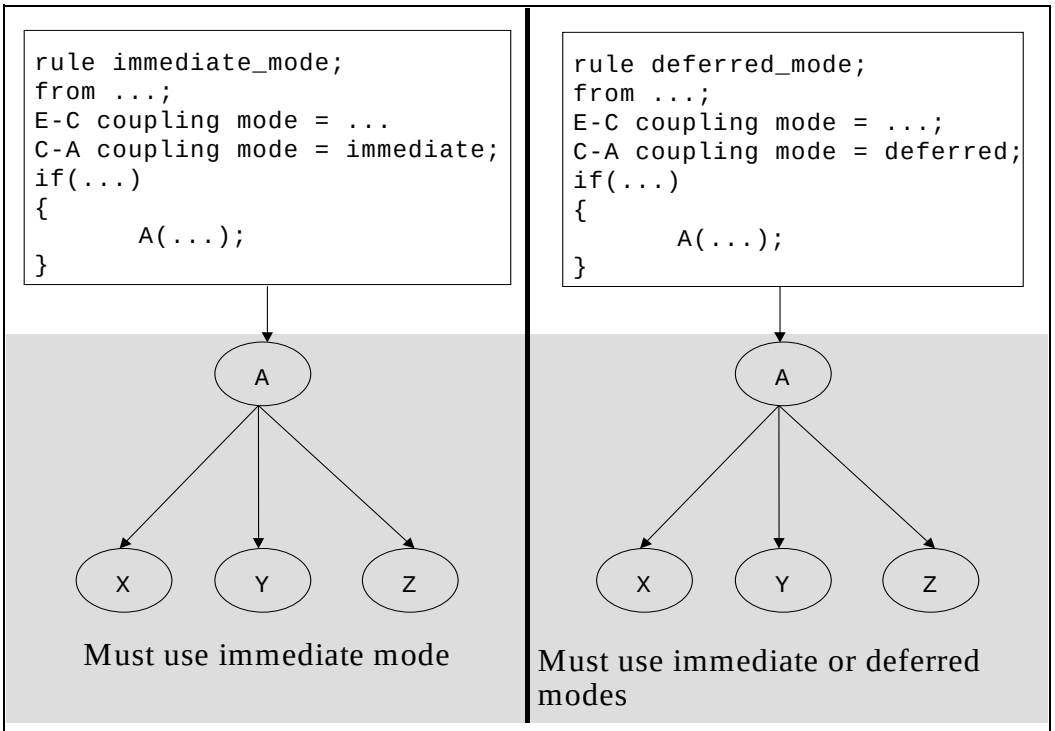


FIGURE 18. VenusDB Coupling Mode Restrictions

- C-A immediate restriction - Only immediate coupling mode is allowed for rule constituents in child modules that can be called from a guard rule stated in C-A immediate mode. This ensures that the child modules will execute in sequential sibling transactions as suggested by the C-A immediate mode.

Figure 18 illustrates these restrictions. On the left, rule `immediate_mode` states its action in C-A immediate mode. `immediate_mode`'s action lists module A. As a result, the rule constituents in A and all of A's children are restricted to immediate mode. On the right, rule `deferred_mode` states its action in C-A deferred mode. `deferred_mode`'s action also lists module A. In this case, the rule constituents in A and all of A's children are restricted to immediate and/or deferred modes.

Lastly, VenusDB's layered architecture does not permit coupling modes. Correl and Miranker have studied this shortcoming [30]. Their solution attaches isolation specifications to individual modules and tables. Three categories of data isolation are proposed called *guard stability*, *serializable*, and *exclusive*. Guard stability mode allows the greatest amount of concurrency, but provides the least amount of isolation from other users. This mode dictates that, at minimum, a row accessed during condition evaluation will be available during action execution. Guard stability mode is the default specification for VenusDB rules. Serializable mode, with properties in between guard stability and exclusive modes, dictates that all rules in nested modules execute within the same transaction. Exclusive mode, being the strongest data isolation mode, dictates that accesses to specified tables are mutually exclusive.

Within LMAs, these isolation modes can be directly mapped to the restricted coupling mode assignments for VenusDB programs presented above. These mappings are:

- Specifying rule constituents in decoupled mode are equivalent to guard stability. In LMAs, rows accessed during condition evaluation are always available during action execution.

- Specifying a rule action in C-A deferred mode under the C-A deferred restrictions described above, is equivalent to issuing serializable mode to the rule's action. This dictates that the rule's action and it's child modules will execute within the same transaction.

- Specifying rule constituents in immediate coupling mode, under the C-A immediate mode restrictions described above, are equivalent to issuing exclusive mode on all tables accessed by the rule. This ensures that data access from immediate rules are mutually exclusive.

- Specifying a rule in E-C deferred mode in which any rule within its module may be triggered by an external event (as opposed to the firing of a guard rule) is equivalent to issuing exclusive mode on all tables accessed in the E-C deferred mode rule's condition. This is stronger than guaranteeing that the condition is evaluated in the same transaction as the triggering external event[12].

4.10.2 Coupling Mode Assignment Algorithm

The definitions presented in this section combined with the techniques of this chapter give rise to the following algorithm for specifying concurrency schemes for VenusDB programs that obey the LMA restrictions.

1. Build a bipartite rule graph for every module as described in Section 4.7.

2. Specify the coupling modes for every rule in every module according to Theorem 8.

3. For every guard rule that uses C-A deferred mode, change all decoupled mode assignments in all rules in all child modules to deferred mode.

   Step 3 maintains correctness since it only restricts the concurrency scheme proposed by Theorem 8.

4. For every guard rule that uses C-A immediate mode, change all coupling mode assignments in all rules in all child modules to immediate mode.

   Similarly, Step 4 also maintains correctness since it only further restricts the concurrency scheme proposed by Theorem 8.

12. This restriction is necessary since Correl and Miranker did not provide primitives for coupling external events to conditions.

5. For every rule stated in C-A deferred mode, surround the entire action in serializable mode. Remove the C-A deferred mode assignments.

   Step 5 maintains correctness by applying the mapping of C-A deferred mode to VenusDB primitives as described above. Further note that Step 3 changed all decoupled mode assignments in all child modules to deferred mode. Thus, this step need not be recursive.

6. For every rule that uses immediate coupling mode, issue exclusive mode on all tables accessed. Remove the immediate coupling mode assignments.

   Similarly, Step 6 maintains correctness by applying the mapping of immediate mode to VenusDB primitives as described above. Further note that Step 4 modified all coupling modes of all child modules from a guard rule stated in C-A immediate mode to immediate mode. Thus, this step also need not be recursive.

7. For every rule that is stated in E-C deferred mode in which any rule within its module may be triggered by an external event, issue exclusive mode on all tables accessed by its condition. Remove the remaining E-C deferred mode assignments.

   Similarly, Step 7 maintains correctness by applying the mapping of E-C deferred mode rules that may be triggered by external events to VenusDB primitives as described above.

8. Remove all remaining coupling mode assignments.

   Step 8 maintains correctness since the only remaining coupling modes are decoupled. In VenusDB, the default isolation primitive is serializable mode which is equivalent to stating LMA rules in either or both of E-C and C-A decoupled modes (as described above).

4.11 Conclusion and Future Work

A large number of coupling modes have been developed to provide the flexibility to efficiently integrate active rules within database transactions. However, coupling modes become unmanageable within hard active database applications. Towards this end, this chapter presented formal execution semantics and correctness proofs for concurrency schemes for a significant subclass of hard rule systems called Log Monitoring Applications (LMAs). LMAs are expert system applications that analyze logs maintained in a database. The concurrency proofs demonstrate that the number of applicable coupling modes are significantly reduced for programs obeying the LMA restrictions. Specifically, the first set of proofs establish that LMA^+ programs, LMAs with only positive variables, are confluent, and thus, their rules can use the most flexible coupling modes of E-C and C-A decoupled modes. Since decoupled modes maximize concurrency, it is fair to conclude that DBMS's need to only support decoupled coupling modes in order to support the execution of LMA^+ programs. The second set of proofs establish that for LMA^- programs, LMAs with both positive and negated variables, only one rule in a set of conflicting rules must be made atomic using E-C and C-A immediate coupling modes. However, the remaining rules may use more flexible coupling modes. Thus, the resulting concurrency control schemes minimize the coupling mode support necessary for the underlying database.

A separate contribution of this work is that the constructive proof techniques can be exploited to build an algorithm for implementing concurrency schemes. Such an algorithm insulates application programmers from the complexities of coupling modes. This algorithm would operate by constructing a rule dependency graph from an input rule program. The algorithm then walks the

graph and outputs the concurrency schemes presented in this paper. Section 4.10 details an example of this algorithm within the VenusDB compiler.

Lastly, algorithms such as the ones presented in Section 4.10 combined with the techniques of this chapter represent the first step in building a general purpose rule compiler that completely isolates application programers from integrating rules within a database. It is the author's belief that the details of transaction models and concurrency schemes are application dependent. Therefore, we expect similar studies to be performed on other application classes. As the number of investigated problem areas are expanded, this compiler will be presented 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.

Chapter 5 The VenusDB Optimizer

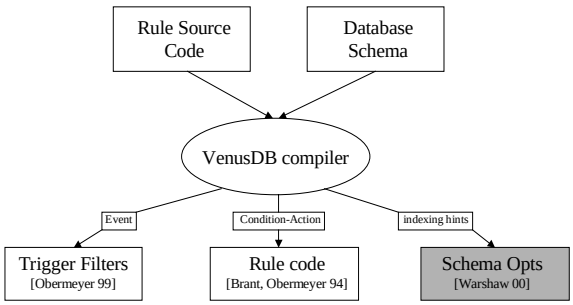


FIGURE 19. VenusDB Compiler Architecture

Rule-based applications are computationally intensive. As a result, one of the main themes of rule-based language research has always been performance. Within hard active databases, performance issues are magnified since rule evaluation spawns queries over the contents of a database. Consequently, inadequate performance increases the duration of locks inhibiting overall system throughput. Towards this end, a primary concern in the development of VenusDB has been performance. As such, VenusDB is a compiled active database language (Figure 19). The compiler was originally designed to take as input rule language source and database schema information [30]. The output was to be a set of trigger filters implementing the event mechanism [69], optimized C++ rule code that tightly integrates the LEAPS match algorithm via the AMI [16,41,60,69], and a set of schema optimizations. This chapter introduces the VenusDB optimizer, an optimizer guided by component database statistics to suggest physical schema optimizations.

The VenusDB optimizer differs from current state-of-the-art active database optimizers that solely rely on the underlying database optimizer [74,98]. A limitation of the database optimizer is that it only optimizes single rules at execution time. As a result, the underlying database schemas are deemed constant. In hard active databases, large quantities of rules may request queries using common predicates. Indexing provides tremendous performance improvements in such an

environment [16,56,69]. Complicating matters is that the choice of indices may be quite daunting. Network administrators must make a greedy choice among a combinatoric number of possibilities. The VenusDB optimizer assists administrators in this task. In addition, based on its physical schema suggestions, the optimizer determines the set of queries to be sent through the AMI for execution on the component databases. The resulting system reduces the deployment costs for scalable active database programs.

This chapter begins by introducing background material, it then details the architecture of the VenusDB optimizer, an instance of an extensible rule-based optimizer written in Venus [92]. Lastly, it concludes with an empirical evaluation.

5.1 Background

Performance tuning is a requirement of all enterprise database applications. Related administration tasks include table normalization, distribution, and replication of data, as well as index selection. Index selection, however, has long proven to be one of the most difficult, but most effective, performance related tasks. Administrators must be keenly aware of workloads, data distributions, table size, and other database resource issues in order to implement useful indices.

Several attempts have been made in automating index selection [28,51]. Common elements of these solutions are providing accurate workloads and methods for pruning the large space. One particularly interesting approach was proposed by the AutoAdmin project at Microsoft Research [25,26]. In a pair of publications, Chaudhuri and Narasayya present a suite of tools that suggest and analyze the impact of indices within Microsoft SQL Server. The tools use the syntax of queries to eliminate obvious inappropriate candidates. Search proceeds iteratively by first optimizing individual rules for single column indices. Search

continues by next optimizing workloads using multiple key indices. A unique characteristic of their work is that their tools uses cost estimations incurred from hypothetical workloads that resemble the cost model exploited by SQL Server's query optimizer. This feature minimizes the impact of index optimization to the online system.

The VenusDB optimizer builds on the fundamentals of Chaudhuri and Narasayya's work. It is similar in that it is uses cost estimations to guide its multi-stage search algorithm for index suggestions. However, it differs in that its architecture is rule-based and incurs the benefits described in [92]. Additionally, the VenusDB optimizer specifically addresses the requirements of active databases including join order considerations.

As has been discussed, hard active database applications may spawn large quantities of queries. Optimizing these queries are complicated within VenusDB due to its layered architecture and match algorithm. In this case, declarative language abstractions obscure the actual queries that execute on component databases. The VenusDB optimizer is designed to cope with these issues. Being part of the VenusDB compiler, it is tightly coupled with the target format of its rules and recommends index suggestions accordingly. In fact, it is the optimizer itself that determines the set of queries that are sent to the component databases. This chapter demonstrates the effectiveness of this architecture.

5.2 Architecture

VenusDB is a layered active database. Therefore, the database is treated as a black box that is only accessed through standard database facilities (such as SQL queries). VenusDB implements this communication layer through instances of the

AMI (Section 3.2.2). In general, layered systems often sacrifice speed for flexibility, and VenusDB is no exception.

The VenusDB optimizer facilitates the efficient implementation of VenusDB's layered architecture. It accomplishes this in two ways. First, VenusDB rules can be viewed as statements upon a federated database. Consistent with most loosely coupled federated databases, the VenusDB optimizer decomposes these federated statements into one or more statements that can be locally executed on its constituent datastores. We call this type of optimization *predicate pushdown* - predicates that are pushed to the local database for execution. Second, the optimizer suggests indices that support predicate pushdown. The combination of these techniques reduce the workload of the VenusDB inference engine.

The VenusDB optimizer is implemented using the extensible Venus-based optimizer architecture detailed in [92]. As such, the VenusDB optimizer defines the operator tree and cost model in C++, and its space of algebraic rewrites and search strategy in Venus. Among the many benefits of this architecture, the cost model uses database statistics. Therefore, the search for optimal indices is guided by actual database workloads.

5.2.1 Optimization Suite

This section details the VenusDB optimizer's optimization suite consisting of predicate pushdown and schema suggestions. To begin, the VenusDB match algorithm must be reviewed as it applies to the suite.

5.2.1.1 LEAPS Algorithm

Venus Rule

```
rule r;
from X[?] x;
     Y[?] y;
     Z[?] z;
if(...) {
  ...
}
```

C++ Implementation

```
void evaluate_rule_r() {

  Cursor X = Container_X.newCursor();
  Cursor Y = Container_Y.newCursor();
  Cursor Z = Container_Z.newCursor();

  for(X.reset();!X.atEnd();X.next()) {
  // predicate over X's value

  for(Y.reset();!Y.atEnd();Y.next()) {
  // predicate over X and Y's values

  for(Z.reset();!Z.atEnd();Z.next()) {
  // predicate over X,Y,and Z's values
  {
    performAction();
  }}}}
```

FIGURE 20. Example Rule and Its C++ Implementation

VenusDB implements rule matching using the LEAPS algorithm. This section summarizes the necessary elements of this algorithm for understanding VenusDB's optimization suite. For a complete description, refer to [16,17,66].

The LEAPS match algorithm is an implementation of the so called *match-select-act* cycle. In the match-select-act cycle, rule evaluation begins in the match phase which evaluates all of the rule guards against the contents of the database and produces a *conflict-set* of satisfied rules. The individual database elements that satisfy a particular rule guard is called an *instantiation*. Next, the select phase picks a single instantiation from the conflict set to pass to the act phase. Lastly, the act phase executes the rule action, and the rule is said to have *fired*. This cycle continues until no further rules satisfy the match cycle. This state is called *fixed point*.

The LEAPS algorithm, as opposed to the other common match algorithms [39,63], is lazily evaluated. Lazy evaluation differs from eager evaluation in that the match phase produces only a single instantiation to pass to the select phase. Therefore, search in the LEAPS algorithm is the process of finding a single instantiation in the match phase. If no instantiation is found, fixed point is reached and the algorithm terminates. Otherwise, the match-select-act cycle continues as normal.

The search for a valid instantiation within the LEAPS match phase is essentially implemented using nested loops for each rule. For example, consider Figure 20. The figure presents a sample VenusDB rule and an abstraction of its C++ implementation. Rule `r` is a 3-way join. The implementation of rule `r`'s condition is a triply nested loop, one for each existential quantifier[1]. `evaluate_rule_r` begins by declaring three cursors over the containers to be evaluated. Each loop then uses the cursors to traverse the containers while testing predicates against the cursor values. If a predicate is satisfied, the next inner loop will be executed. If a predicate is not satisfied, the cursor is moved forward. When the innermost loop is reached, the rule action is performed[2].

The LEAPS algorithm continues evaluating until fixed point is reached.

5.2.1.2 Predicate Pushdown

1. This is actually inaccurate. Rule `r` would really be implemented using only two loops. In LEAPS, search is seeded by search points that reduce the join-arity by one. The details of this mechanism is beyond the scope of this section.
2. Though not illustrated, the search for instantiations is suspended during action execution. After the action completes, search is resumed, not necessarily in the same rule, based on a priority queue of search points. This queue is the essence of the LEAPS algorithm.

A close inspection of Figure 20 reveals wasted work. In Venus, containers and their elements live in main memory. Therefore, iterating over the values of containers and testing one at a time is feasible. Conversely, VenusDB containers and elements may live over a network on a database. Nested loop evaluation in this context is equivalent to locally materializing the contents of an exponential number of "`select *`" queries per rule evaluation without regard to indices. Further, recall that LEAPS is a lazy match algorithm. Therefore, the generation of the entire stream of data is rarely exhausted since only one instantiation is produced on each cycle.

Figure 21 presents the same rule presented in Figure 20 with an alternate C++ implementation using predicate pushdown. In this version, evaluation of rules still occurs using nested loops. However, the rule predicates are removed from the loops, parameterized, and inserted into the `newCursor` methods[3]. The `reset` methods substitute the actual values of the cursors in the predicates. Before iteration, the implementing streams (AMI implementations) relay these predicates to their local databases for execution. Therefore, instead of executing full relation scans, the database executes finely constrained select predicates that take advantage of querying and indexing utilities. If a query returns an empty result, the loop is aborted. Otherwise, the loop is traversed the same as before, but hopefully with a substantially reduced stream size.

This optimization facility is modeled after the cursor creation scheme in Oracle and other relational database management systems. In these systems, the creation of a result set for a query statement is a multiple step process. This pro-

3. In reality, the predicates are repeated within the loops as in Figure 20. This allows the VenusDB engine to correctly process predicates if predicate pushdown is not supported by an AMI implementation.

Venus Rule

```
rule r;
from X[?] x;
     Y[?] y;
     Z[?] z;
if(...) {
  ...
}
```

C++ Implementation with Predicate Pushdown

```
void evaluate_rule_r() {

  Cursor X = Container_X.newCursor(
      /* predicate over X's value */);
  Cursor Y = Container_Y.newCursor(
      /* predicate over X and Y's
         values */);
  Cursor Z = Container_Z.newCursor(
      /* predicate over X,Y, and Z's
         values */);

  for(X.reset(/* parameter subst */);
      !X.atEnd();X.next()) {

  for(Y.reset(/* parameter subst */);
      !Y.atEnd();Y.next()) {

  for(Z.reset(/* parameter subst */);
      !Z.atEnd();Z.next()) {
  {
    performAction();
  }}}}
```

FIGURE 21. Example Rule and Its C++ Implementation using Predicate Pushdown

cess begins with the parsing of an SQL query to build an optimized execution plan. For efficiency, database systems usually allow the query to include parameters. Thus, a single SQL query is parsed and optimized once, but executed multiple times. Parameterization is achieved by embedding formal parameters within the SQL query. To retrieve rows, a query execute function is called. For queries that have formal parameters, actual parameters are substituted. For example, a valid SQL query for Oracle is "`select a, b from c where a = :1`." In this statement, "`:1`" is a formal parameter. The call to the execute function will contain an actual parameter to substitute for the formal parameter.

Similarly, when using predicate pushdown, the `newCursor` statement includes a parameterized predicate. This is a logical statement that contains constant tests and formal parameters for substitution. For example, "`x == 1 && y = :1`" is a valid parameterized predicate. Before iterating over a container, a cursor must call the reset method. If the cursor was created using a parameterized predicate with formal parameters, the reset method substitutes the values of the current cursor positions as actual parameters.

The results of these definitions yield a flexible optimization scheme. If a component database supports an advanced query capability, predicate pushdown may result in significant performance benefit. Otherwise, the predicate statements are ignored, and full relation scans occur.

5.2.1.3 Indexing

The scalability of rule programs have been proven to improve when using indices [69]. Thus, it follows that the scalability of VenusDB programs will also improve with the use of indices.

However, determining the optimal indices for the complex applications this dissertation addresses can be quite difficult. In these cases, database administrators must identify indices from the large space consisting of the number of predicates in the rule programs times the aggregate number of columns of all exploited database schemas. Further, the administrators must use workflow information to decide which queries are more likely to be executed than other queries, and which of these queries deserve optimization. Lastly, the previous section demonstrated that the VenusDB match algorithm may spawn somewhat unexpected query statements.

The VenusDB optimizer is designed to assists database administrators in their schema optimizations. The optimizer accomplishes this task by using database workloads to suggest indices that directly support the database queries spawned by the LEAPS algorithm.

5.2.2 Rule-based Implementation

The VenusDB optimizer is an instance of the Venus-based optimizer, an extensible rule-based optimizer written in Venus [92]. The adaptation of this extensible architecture has tremendously reduced the development effort of the VenusDB optimizer. Features of the Venus-based optimizer that have proven ideal for this project include:

- The exploitation of Venus' modularity to encapsulate the optimizer components of rewrite and search as a set of declaratively expressed rule modules without sacrificing performance.
- A well structured rule environment that operates within fixed-point semantics [18,19].
- Embeddability in C++ and thus embeddability within VenusDB. Venus' data definition language is precisely C++. Thus, benefits seen in extensible object-oriented optimizers developed by [57,72] with respect to the operator tree and cost model are exploited identically.
- An optimizing compiler [64,91].
- A familiar C++ syntax.

This section introduces the VenusDB instance of the Venus-based optimizer.

5.2.2.1 Optimizer Components

The four basic parts of an optimizer consist of the *operator tree*, *cost model*, *search-space* and *search strategy.* The operator tree is the machine representation of a statement, and uses the cost model to estimate the efficiency of the statement. The search space is defined as the space of algebraically equivalent transformations defined in the rewrite system, while the search strategy uses the cost model to search for an optimal plan within the search space. The Venus-based optimizer encapsulates each of these components.

5.2.2.2 Data Model

Due to its connection to C++, the Venus-based optimizer's operator tree and its associated cost model are defined in terms of first-class C++ objects that closely resemble the object-oriented operator tree definitions developed in [57,72] (Figure 22). The rule-system exploits this API which in turn decouples the rule-system from the data model.

5.2.2.2.1 Operator Tree

The operator tree is implemented using an abstract class hierarchy that represents algebraic database operators combined with the abstract methods and attributes needed to define a *descriptor*, the logical and physical description of the operator [32]. Following conventional definitions, a *logical operator* is defined as an algebraic operation that operates on its inputs, and a *physical operator* is defined as a logical operation that has been assigned an implementing algorithm.

Figure 22 presents a UML class diagram for the VenusDB operator tree. The shaded area illustrates the classes that are predefined by the Venus-based optimizer. The base class, `OPERATOR`, represents an algebraic operation. The classes

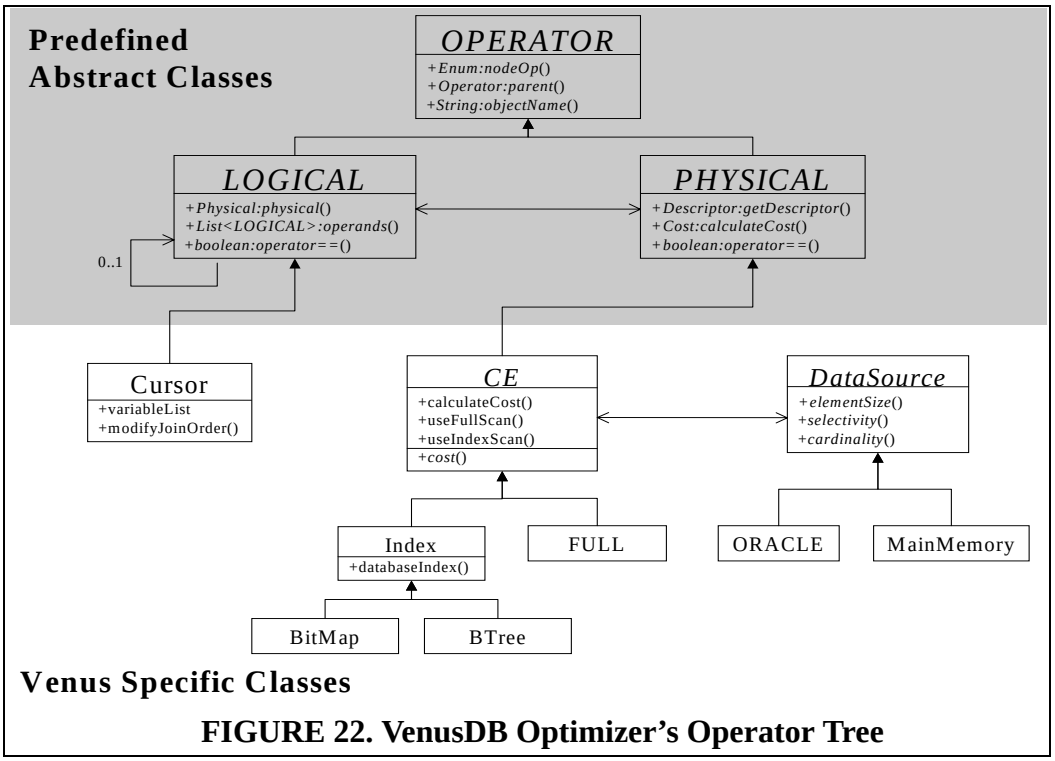


FIGURE 22. VenusDB Optimizer's Operator Tree

`LOGICAL` and `PHYSICAL` are derived from `OPERATOR`, representing a logical and physical operator respectively, and contain a pointer to one another. The spine of the operator tree is maintained within the logical operator. As studied in the Volcano and OPT++ efforts, this separation of algebra has proven effective [48,57].

The non-shaded area of Figure 22 illustrates the classes implemented for the VenusDB optimizer. Recall, VenusDB rules are evaluated using nested loops. Each loop ranges over a container with either an existential or universal cursor (Sections 3.1 and 5.2.1.1). Thus, the VenusDB operator tree specializes the Venus-based optimizer's class hierarchy by representing rule conditions with an ordered list of cursors. The tail of the list represents a (possibly non-restrictive) cursor predicate over a container. Internal nodes represents a cursor predicated as well as a nested loop join with the next element in the list. The list is ordered from outer-most to innermost loop.

Towards this end, the `Cursor` class encodes the logical representation of a cursor. A `Cursor` contains a maximum of one operand. Additionally, `Cursor` maintains a list of variables in its predicate that must be "bound" before loop evaluation. A variable is bound when it is assigned a value during loop iteration. In other words, consider the following rule:

```
rule A;
form X[?] x;
     Y[?] y;
if(y.a == 3 && x.a == y.a) {..}
```

If the cursor `y` is used as the outer loop, then `y.a` attribute will be bound to a value, specifically the number `3`, during loop iteration. Later, during `x`'s loop iteration, `x.a == y.a` will use `y.a`'s bound value, the number `3`, to evaluate the predicate. Though not illustrated, cyclic predicates constrain the possible ordering of nested loops. The `Cursor`'s `variableList` maintains this information.

The physical representation of a cursor is encoded within the `CE` class[4]. `CE` is an abstract class that represents the type of database retrieval that will be used for cursor evaluation. Consequently, the `Index` and `Full` classes are derived from `CE` to represent index and full table scans respectively. The `Index` class is further refined with the `BitMap` and `BTree` classes to represent bitmap and B-tree indices respectively.

`CE` also contains a pointer to its originating data source that is used for retrieving descriptor information. This is represented by the abstract `DataSource` class. The VenusDB optimizer described in this section ranges over main

4. CE stands for conditional element. Cursors are called CEs in Venus's implementation code.

memory and Oracle *8i* AMI instances. Therefore, `ORACLE` and `MainMemory` classes are derived from `DataSource`.

5.2.2.2.2 Cost Model

The cost model within the Venus-based optimizer is defined in terms of abstract methods associated with the physical operators (Figure 22) . Each physical operator is refined to contain a set of implementing algorithms. It is these implementing algorithms that refine the `calculateCost` method which is used by the rule system to calculate a cost of an algebraic operation.

The VenusDB optimizer exploits the Venus-based cost model in order to use database statistics. This is accomplished by abstractly defining the cost within the `CE` class. In turn, the `CE` class implements the `calculateCost` method through an abstract interface. This design allows the descriptor information to be filled by actual database statistics.

The cost model of a VenusDB rule closely resembles the cost models for standard SQL queries presented in [10,54]. The cost of a rule is composed of the costs of its individual cursors. The cost of a cursor is dependent on both its restricting predicate and its access path. For an existential cursor ranging over the container *A* and predicate *P*, the cost of the cursor is calculated by

$$\text{cost}(A,P) = \begin{cases} \text{card}(A) & \text{if A is a Full scan} \\ \log(\text{card}(A)) * \text{select}(A,P) & \text{if A is a B-Tree scan} \\ \text{select}(A,P) & \text{if A is a bitmap scan} \\ \text{card}(A) * \text{select}(A,P) & \text{if A is a Predicate scan} \\ \text{card}(A)/1000 & \text{if A is in main memory} \end{cases} \quad \text{(EQ 1)}$$

where *card*(*A*) is the cardinality of *A* and *select*(*A,P*) is the selectivity of *A* as restricted by predicate *P.* [5]

The selectivity of a cursor is dependent on the range of the values of the container it will traverse and the predicate it will execute. VenusDB calculates the selectivity of a cursor *A* as restricted by predicate *P* by

$$\text{select}(A,P) = \begin{cases} \text{catalog estimation}, & \text{if } P \in \{==\} \\ \text{1- equality estimation}, & \text{if } P \in \{!=\} \\ 0.50, & \text{if } P \in \{>,>=,<,<=\} \end{cases} \quad \text{(EQ 2)}$$

The catalog estimation is the primary utility within the VenusDB optimizer to use workload information. The catalog estimation of main memory containers are roughly estimated. However, predicates ranging over database containers exploit database statistics. In particular, depending on availability, the `ORACLE` class estimates selectivity using histogram estimations or the number of elements within leaves of an index. Primary and foreign key information is also exploited.

Predicates may be grouped with disjunctive and conjunctive logical connectives. As such, the selectivity for logical connectives are computed by

$$\text{select}(A \bullet B) = \begin{cases} \text{select}(A) * \text{select}(B) & \text{if } \bullet \text{ is \&\&} \\ 1 - (1-\text{select}(A)) * (1 - \text{select}(B)) & \text{if } \bullet \text{ is } \| \end{cases} \quad \text{(EQ 3)}$$

Lastly, the estimated cost of a rule is calculated recursively. For a rule that contains only a single cursor, the cost of the rule is simply the cost of the cursor. Rules composed many cursors form nested loop joins. The calculated cost of joining cursors *A* and *B* where *A* is the outer loop and *B* is the inner loop is

5. Universal cursors are implemented by first applying DeMorgan's law. The cursor evaluation then fails if any values are returned (Venus uses the closed-world assumption). Therefore, the estimated cost of an universal cursor is equivalent to an existential cursor with a selectivity equal to one.

join(A,B) = cost(A) + select(A) * card(A) * cost(B) (EQ 4)

The total estimated cost of a VenusDB rule equals the estimated cost of the outermost join.

5.2.2.3 Rule System Architecture

The rewrite and search components are segregated in separate modules and are implemented using the Venus rule language.

The module call graph of the rule components is illustrated in Figure 23. The Venus-based optimizer is cost-driven. Therefore, search strategies use cost estimates to sequence and prune the search space.

5.2.2.3.1 Rewrite System

The search space is defined by the space of algebraic rewrites presented in the rewrite system. Like the previous generation of rule-based optimizers, the rewrite system in the Venus-based optimizer is implemented in rule-based form [32,47,48,80]. The rewrite system applies different transformation and implementation rewrites on a subquery by exploiting procedural and heuristic elements, constrained by the algebraic representation. If the preconditions and conditions for a rewrite are satisfied, the operator is applied and the new subquery is passed to the post-conditions modules which update accordingly.

The VenusDB optimizer contains three rewrite rules.

The `modifyJoinOrder` rule reverses the order of nested loops[6]. This rule additionally has the effect of granting the VenusDB optimizer control over the

6. Note, changing the order of a nested loop is only sometimes equivalent to a join commute. In some cases, the operation is equivalent to applying a sequence of commutative and transitive operators.

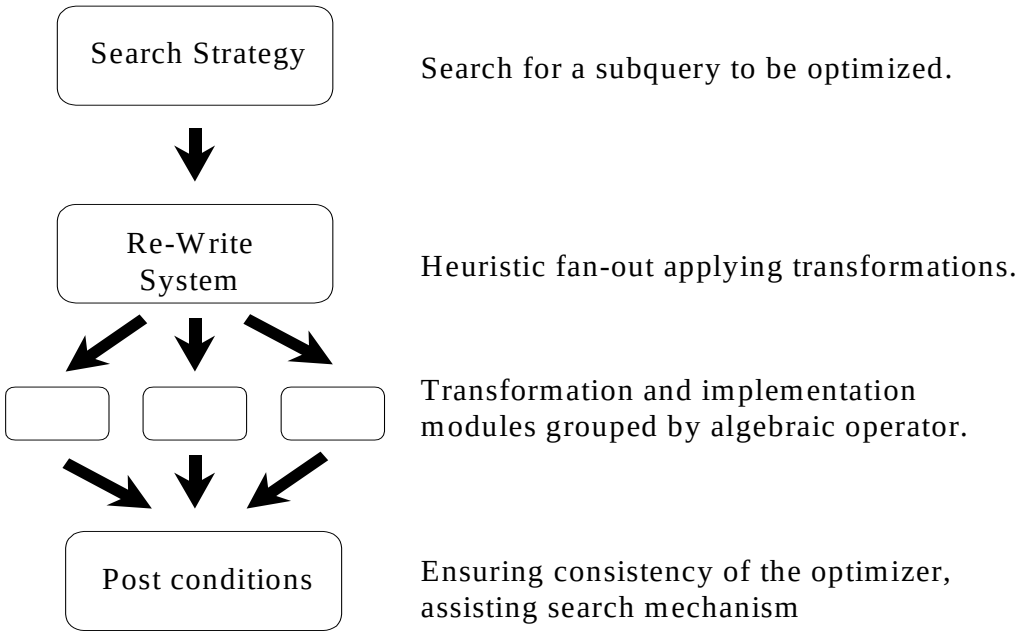


FIGURE 23. Rule System Architecture

predicates that will be executed on component databases. Due to bound-variable restrictions, this rewrite must validate the legality of loop reordering by checking the cursor's respective `variableList`'s.

The other two rewrite rules, `useIndexScan` and `useFullScan`, change the access paths of cursors. `useIndexScan` assigns an index to a cursor. This rule chooses its index selection from a table that is computed prior to the rewrite stage. If the index to be created is on a single column and the number of distinct values within the column is less than the log of the cardinality of the table, a bitmap will be used. Otherwise, a B-tree will be used. `useFullScan` assigns a full table scan to a cursor. There are no restrictions in the application of `use-FullScan`.

5.2.2.3.2 Search Strategy

The VenusDB optimizer's search strategy occurs in three phases. In the first phase, a graph rewriting algorithm greedily prunes the number of useful indices. This algorithm begins by inspecting every rule for cursors that are involved in a non-inequality predicate. Each such predicate is considered a potential index. These indices form the nodes of a graph. Additionally, the nodes are adorned with the number of cursors that may use the index. The algorithm then inspects rules for potential multiple column indices. Candidates include cursors involved in multiple predicates within the same rule. Arcs are drawn connecting nodes accordingly. The algorithm completes by building an index table that contains all nodes within the graph as well as all of the multiple column indices described by connected components. Since data locality affects the usefulness of an index, the multiple column indices are ordered from the nodes with the most predicates to the least predicates.

The second phase of search optimizes VenusDB rules using the Venus-based optimizer's search template to implement hill climbing. This template is modeled after classic AI heuristic search and is implemented using the Venus rule language.

Heuristic search is often represented as a directed graph rooted from a designated start node. Nodes within the graph represent states in the search space. An arc connecting a pair of nodes represents an application of an operator by the search routine. Search is the process of traversing the graph through the expansion of nodes by operator applications in an attempt to find a goal criterion. The encoding of search is commonly represented with an open and closed list. The open list contains nodes that have been investigated but not expanded by the search routine, and the closed list contains nodes that have been expanded by the search routine [77].

Following this model, the top-level module(s) coupled with the post-condition module(s) of the Venus-based optimizer define the search strategy. The top-level module(s), represented as the search strategy box in Figure 23, choose a subquery to pass to the rewrite system based on cost estimates, while the post-conditions module(s) update according to search strategy.

Implementing hill climbing is straight-forward using this template. Hill climbing is a simple search strategy that applies all possible rewrites to a query and retains only the least cost rewrite. The search continues by recursively rewriting the least cost rewrite and completes when the rewriting yields no improvement.

Consequently, the encoding of hill climbing roots the search graph with the rule to be optimized. The open list contains each cursor within the rule. The closed list stores a log of all expanded rewrites. The search method modules pick a least-cost cursor from the open list. The rewrite system expands the cursor using rewrite rules, while the post-condition module(s) places only the most improved cursor in the open list and moves all other cursors to the closed list. Optimization continues in this fashion until rewrites no longer provide cost improvements.

In this way, rules are rewritten within the space of VenusDB rewrites. Joins are reordered and indices are suggested from the index table. Cost is estimated using the VenusDB cost model. When all rules have been optimized, the set of indices used in optimal rule configurations are gathered together in an index suggestion table.

The third phase of search prunes the index suggestion table by eliminating replicated indices and aggregating other indices using heuristics. Currently, the primary heuristic used is to eliminate a multiple column index when another multiple column index in the table contains a superset of its columns.

Lastly, the index suggestion table is reported. The report includes the rules that benefit from each index, the estimated performance improvement, and the estimated size of the index.

5.2.3 Use of the VenusDB Optimizer

Database administrators should employ the following algorithm when using the VenusDB optimizer.

1. A rule program is fed to the VenusDB compiler. The compiler outputs rule code as well as index suggestions.

2. The suggested indices are created on the component databases.

3. The rule program is again fed to the VenusDB compiler. The compiler will produce rule code as well as the VenusDB optimizer's index suggestions. These index suggestions may be different than the previous suggestions due to the statistic estimations gathered from the component databases descriptors.

4. The index suggestions from the previous two compiler executions are compared. Indices that are no longer recommend are deleted. New index suggestions are created on the component databases.

5. The database administrator then repeats Steps 1-4 as deemed appropriate.

5.3 Empirical Evaluation

The goal of the empirical evaluation is to demonstrate that the optimizer:

- Improves overall performance for rule programs that vary with the complexity of search and data. This demonstrates the overall scalability improvement gained by the VenusDB optimizer.

Program	Rules	Cond/Rule	Columns	Description
Waltz	25	6.3	12	3D line labeling through constraint satisfaction
Manners	9	3.1	16	Arranges seating of guests using depth-first search
TSP	24	10.8	60	Traveling salesman problem using greedy heuristics
TPC-D	3	10	61	Transaction Processing Performance Council query
REALESYS-A	74	3.5	15	Active database mortgage pool allocation program

TABLE 7. Summary of Test Programs

- Reduces the time to evaluate each rule. This measures the amount of time the database may be locked during rule evaluation. A reduction means that the transaction time has been decreased and is particularly important in reducing the lock times of "immediate" coupling mode rules (Section 2.2.2).

- Chooses indices that otherwise would not have been obvious. This demonstrates that the optimizer successfully assists database administrators in complex database design issues.

5.3.1 Test Programs

To satisfy the goals of the experiment, the empirical evaluation was performed on five programs from the University of Texas benchmark suite [17,69]. The programs vary with the size of data, complexity of rules, number of rules, and number of attributes within relations in order to measure the scalability improvements afforded by the VenusDB optimizer. A separate requirement of each selection is that the database driving the programs could be atomically generated with control parameters. Table 7 summarizes the following descriptions of the selected test programs.

Waltz

Waltz, developed at Columbia University, is a program designed to map a two-dimensional drawing to a three-dimensional space. The drawing algorithm

uses a constraint satisfaction solver. Constraint satisfaction proceeds by labeling lines in the drawing according to its neighbors. The input to a Waltz program consists of Cartesion coordinates representing lines. The data generator scales according to a base drawing called a *region*. A region contains 72 lines. Waltz contains 25 rules with 6.28 conditions per rule. The aggregate number of all columns in all tables is 12. Waltz is an example of a program with lots of search and moderate data requirements.

Manners

Manners is a depth-first search for seat assignments at a dinner arrangement. Seat assignments are limited by etiquette rules such as neighbors should be of the opposite sex and share hobbies. The number of guests is a parameter of the data generator. Manners contains 9 rules with 3.1 conditions per rule. The aggregate number of all columns in all tables is 16. Manners is another example of a program with lots of search and moderate data requirements.

TSP

TSP is an implementation of the classic traveling salesman problem. TSP uses greedy heuristics to determine the minimum distance route for a salesman to visit all cities in his/her itinerary. The program contains rules that heuristically choose when to travel to new cities and when to cross borders of states. The data includes the Cartesion coordinates of each city as well as a clique describing the distances between each city. A parameter of the data generator is the number of cities to visit. TSP contains 24 rules with 10.83 conditions per rule. The aggregate number of all columns in all tables is 60. TSP is an example of a program with extensive search and data.

TPC-D

The Transaction Processing Performance Council periodically publishes a suite of database benchmarks [89]. TPC-D is one such benchmark that measures performance of databases with typical workloads. The benchmark defines a database schema, a series of queries and a data generator. The size of the data is a parameter of the data generator.

The VenusDB TCP-D program is adopted from [69]. This program models one of the more complex queries within the TPC-D benchmark. Though this program cannot be considered an actual TPC-D result, it is a good example of a program with selective search and lots of data. TPC-D contains 3 rules with 10 conditions per rule. The aggregate number of all columns in all tables is 61. The TPC-D program characteristics are representative of the program described in the Appendix.

REALESYS-A

REALESYS-A is the active database version of REALESYS described in Section 3.3.2.2 [71]. The data generator takes as input the number of pools that have been traded to fill contracts. REALESYS-A contains 74 rules with 3.5 conditions per rule. The aggregate number of all columns in all tables is 15. REALESYS-A is an example of a program with lots of search and little data.

5.3.2 Test Harness

The empirical evaluation was performed on an instance of the VenusDB optimizer that mirrored the architecture described in Section 5.2.2. The optimizer ranges over main memory and Oracle 8*i* containers.

However, due to the realities of Oracle 8*i*, the optimizer contained noticeable modifications from the original design. In Oracle 8*i*, there is no way to ensure that Oracle will use an available index. Oracle does publishes a set of language

statements, called *hint syntax*, that is embedded within queries to provide implementation hints. However, these statements are only hints. Therefore, there are cases when suggested indices are ignored by Oracle's optimizer despite hint syntax. Next, we did not find a published method to interpret Oracle's histogram on data that is neither discrete or numeric. For these reasons, three modifications were made. First, the cost model uses the number of leaves per key of an index for selectivity estimations. If an index is not available, selectivity is calculated based on the predicate type and a constant percentage of the attribute's cardinality. Second, the optimizer augments predicate pushdown statements with Oracle hint syntax. This modification requires an additional step in the VenusDB usage algorithm presented in Section 5.2.3. In the new step, predicates are analyzed using SQL's plan analyzer. Indices ignored by the Oracle optimizer are deleted. Third, the optimizer only modifies join orders in the face of overwhelming evidence. The basic greedy heuristics of the VenusDB compiler proved difficult to improve upon [64].

With the optimizer in place, the empirical evaluation was performed by executing each test program on a series of equally spaced increasing data sizes. At each data size, the program was run three times. The first run did not contain any VenusDB optimizations, i.e., no predicates were pushed to the Oracle database and no indices were implemented; primary keys were established on LEAPS time-tags. The second run turned on the predicate pushdown utility but indices were not created on the Oracle database. The third run utilized both the predicate pushdown utility and the final set of index suggestions from the VenusDB optimizer after completing the algorithm presented in Section 5.2.3. All tests were performed on a Sun Ultra-60 with two 450-MHz UltraSPARC II processors running Solaris 2.7. The machine had 16 gigabytes of main memory and a 30 gigabyte RAID disk array. Sun publishes the SPECInt95 rating for configurations of this machine

ranging from 19.7-32.7. The AMI implementation resided over Oracle version 8*i* database.

The two primary measures of *overall execution time* and *average number of cycles per second* were taken during each execution. Overall execution time measures the raw performance improvements achieved by using the VenusDB optimizer. Cycles per second measures the scalability improvements afforded to each rule by the VenusDB optimizer.

Figures 24-28 reports a summary of the results. In each figure, the tables on the left contain the four column labels of 1) Data Size, 2) No Opt, the non-optimized run, 3) Push, the pushdown enabled run, and 4) Index, the final optimized configuration. Values are reported in seconds. A comparison matrix is presented to the right of each table. The matrix contains the three column labels of 1) Push/No Opt, speedup of the pushdown run over the non-optimized run, 2) Index/Push, speedup of the indexed run over the pushdown run, and 3) Index/No Opt, speedup of the indexed run over the non-optimized run. Values in the matrix for the overall execution time are a percentage of reduction in time. Values in the matrix for cycles per second are a percentage of speedup. In both tables, dashes represent programs that were not completed due to excessive lengths of time. The tables are accompanied by two log-scaled graphs. The graph in the lower left plots the execution time results. The graph in the lower right plots the cycles per second results.

Figure 29, located at the end of the chapter, presents the unabridged results. The tables in this figure contain three new columns. The “Opt 1” column contains the execution times after the first run of the VenusDB optimizer; the “Opt 2” column contains the execution times for the final optimized configuration; and the “Firings” column contains the number of rule firings within a program execution. Lastly, the least squares line for each series is presented.

	Execution Time			Reduction		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
1MB	1040	19	17	98.1%	15.1%	98.4%
10MB	11834	115	59	99.0%	48.7%	99.5%
100MB	-	1130	488	-	56.8%	-
1GB	-	16550	11709	-	29.3%	-

	Cycles Per Second			Speed Up		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
1MB	1.04	55.38	65.2	5239.5%	17.7%	6186.4%
10MB	0.06	6.55	12.77	10150.2%	95.0%	19882.8%
100MB	-	0.64	1.49	-	131.5%	-
1GB	-	0.03	0.04	-	41.3%	-

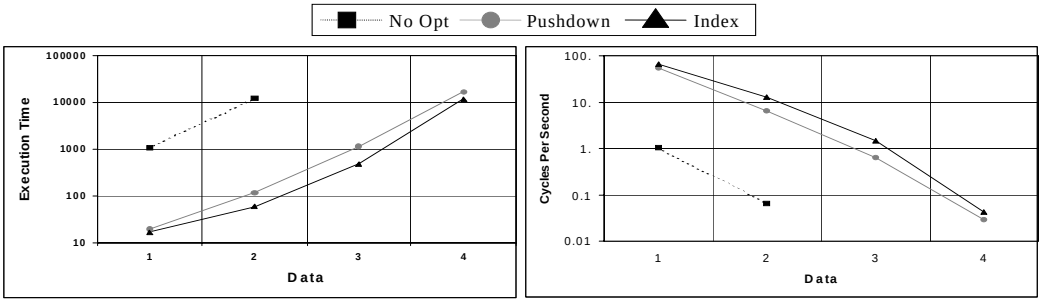


FIGURE 24. TPC-D Results

	Execution Time			Reduction		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
south	30	17	18	42.8%	-2.1%	41.6%
mountains	142	44	43	69.1%	2.0%	69.8%
usa1	423	90	88	78.7%	2.4%	79.2%
usa2	4801	847	674	82.4%	20.4%	86.0%
usa3	17945	3045	2286	83.0%	24.9%	87.3%
usa4	-	6857	4748	-	30.8%	-

	Cycles Per Second			Speed Up		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
south	4.4	7.69	7.53	74.8%	-2.1%	71.2%
mountains	1.92	6.2	6.33	223.9%	2.1%	230.6%
usa1	0.9	4.22	4.33	369.9%	2.5%	381.6%
usa2	0.26	1.49	1.87	466.8%	25.7%	612.3%
usa3	0.14	0.84	1.12	489.3%	33.2%	685.0%
usa4	-	0.59	0.85	-	44.4%	-

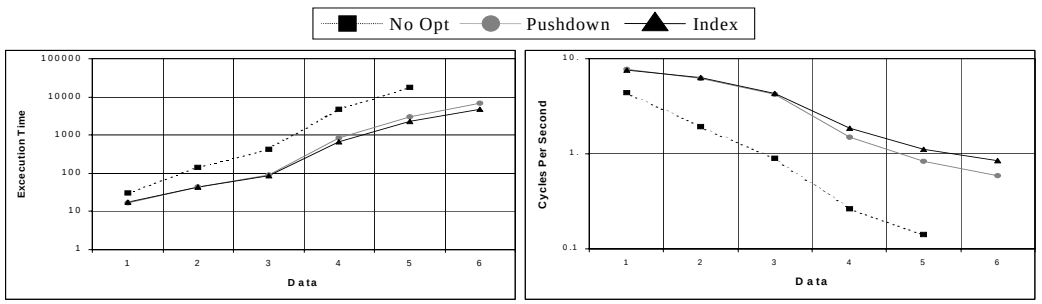


FIGURE 25. TSP Results

	Execution Time			Reduction		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
1	49	48	51	2.0%	-6.3%	-4.1%
2	145	128	136	11.7%	-6.3%	6.2%
4	430	366	365	14.9%	0.3%	15.1%
8	1417	1124	1139	20.7%	-1.3%	19.6%
16	5519	3968	3980	28.1%	-0.3%	27.9%
32	-	17178	17241	-	-0.4%	-

	Cycles Per Second			Speed Up		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
1	5.41	5.52	5.2	2.1%	-5.9%	-3.9%
2	3.73	4.23	3.98	13.3%	-5.9%	6.6%
4	2.54	2.99	2.99	17.5%	0.3%	17.8%
8	1.55	1.95	1.93	26.1%	-1.3%	24.4%
16	0.8	1.11	1.11	39.1%	-0.3%	38.7%
32	-	0.51	0.51	-	-0.4%	-

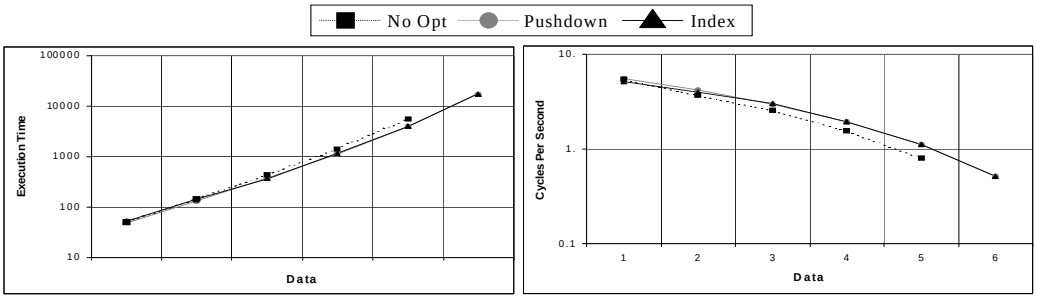


FIGURE 26. Waltz Results

	Execution Time			Reduction		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
16	7	7	6	0.0%	14.3%	14.3%
32	37	23	23	37.8%	0.0%	37.8%
64	345	110	97	68.1%	11.8%	71.9%
128	4500	715	531	84.1%	25.7%	88.2%

	Cycles Per Second			Speed Up		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
16	26.29	26.29	30.67	0.0%	16.7%	16.7%
32	16.86	27.13	27.13	60.9%	0.0%	60.9%
64	6.59	20.65	23.42	213.6%	13.4%	255.7%
128	1.92	12.08	16.27	529.4%	34.7%	747.5%

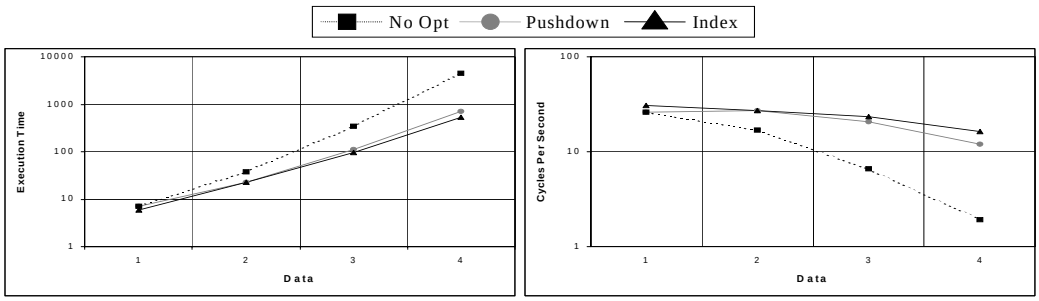


FIGURE 27. Manners Results

	Execution Time			Reduction		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
8	37	35	34	6.1%	2.6%	8.5%
16	67	68	67	-0.4%	0.6%	0.2%
32	247	246	247	0.5%	-0.5%	0.0%
64	523	517	514	1.1%	0.6%	1.7%

	Cycles Per Second			Speed Up		
Data Size	No Opt	Push	Index	Push/No Opt	Index/Push	Index/No Opt
8	29.8	31.72	32.56	6.5%	2.6%	9.3%
16	28.47	28.36	28.53	-0.4%	0.6%	0.2%
32	22.65	22.77	22.66	0.5%	-0.5%	0.0%
64	20.21	20.43	20.56	1.1%	0.6%	1.7%

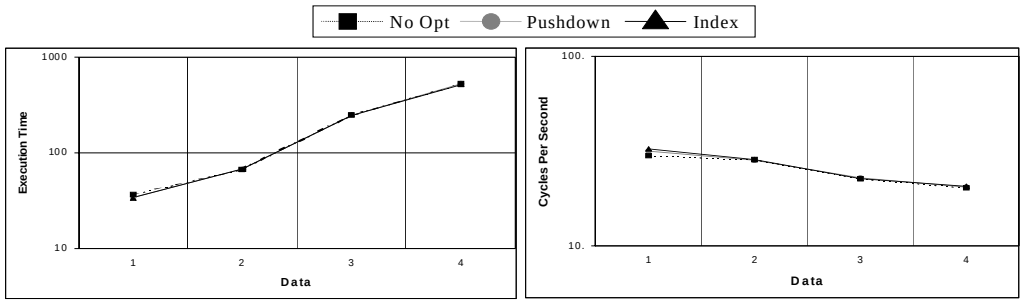


FIGURE 28. REALESYS-A Results

5.3.3 Discussion

The results demonstrate that the VenusDB optimizer succeeds in all three goals of the empirical evaluation: 1) improving overall performance, 2) reducing the time to evaluate rules, and 3) assisting the database administrators with the choice of indices.

The overall execution results demonstrate that both predicate pushdown and indexing are beneficial. With respect to predicate pushdown, the overall performance of TPC-D and TSP programs are improved nearly or greater than an order of magnitude. Manners and Waltz also contain statistically significant improvements, 84% and 39% reductions respectively. The least squares lines further supports the claim that predicate pushdown tremendously improves scalability.

Additional performance improvements are demonstrated with the use of VenusDB's index suggestions in TPC-D, TSP, and Manners. Specifically, large additional performance improvements are demonstrated from the TPC-D and TSP programs, as much as 56% and 30% respectively. These improvements suggest that programs with large amounts of data scale much better with the use of the VenusDB index suggestions. The least squares lines give evidence that such programs can expect nearly a twofold additional speedup. In the case of Manners, program execution is further reduced up to 30%. The Manners plots further suggest that the larger the Manners program, the more benefit gained from indices.

The average cycles per second results magnify the trends established by the overall execution time. In nearly all programs, predicate pushdown reduces average cycle time by several orders of magnitude topped by an astounding 10,150% speedup for TPC-D. Indexing further reduces cycle time in all programs but Waltz and REALESYS-A. In the case of TPC-D and TSP, this additional reduction

proves to quite significant, with speedups up to 130% and 44% respectively. Therefore, it can be concluded that the VenusDB optimizer successfully reduces the cycle time for VenusDB programs. Thus, programs that necessitate immediate coupling mode rules as discussed in Chapter 4 must work in conjunction with the VenusDB optimizer for scalable performance.

Although the evidence is subjective, the optimizer's suggestions support a claim that index selection is best served by an automated utility. This is because the suggestions were not always obvious. The foreign keys of all tables were suggested. Yet, in the case of TSP and Waltz, their foreign keys contain two or more columns. Other interesting suggestions include bitmap suggestions on the `status` column of the `Cities` table in TSP and bitmaps suggestions on both `hobby` columns of the Manners' `Guests` and `Chosen` tables.

The results from the Waltz and REALESYS-A programs provided further insight into the usefulness of the VenusDB optimizer. These programs contain little data and lots of search. Their results demonstrate these properties by exhibiting less performance improvements than the other programs. In fact, REALESYS-A remained roughly constant regardless of the optimization scheme. These programs are examples of expert systems that are designed to consume data as soon as it materializes. Therefore, queries over large tables of data are not performed. Programs with these characteristics are sufficiently optimized using the LEAPS algorithm [64]. The final optimization configurations confirm this claim.

5.4 Conclusion

Hard active database programs make extreme resource demands. Such programs spawn large quantities of queries over large database tables. Rule chaining compounds these demands. Since it is often the case that hard active databases

are used to encode real-time decision control systems, these systems must scale. However, the complex nature of these programs makes optimization tasks non-trivial. Thus, a utility must assist database administrators.

The results of the empirical evaluation suggest that the VenusDB optimizer addresses these concerns. Tables 24-28 indicate that the VenusDB optimizer improves the scalability of VenusDB programs. In fact, the programs with larger data sets, TPC-D, TSP, and manners, tend to be improved the most by using the VenusDB optimizer. In these cases, an order of magnitude performance improvement is not uncommon. The index suggestions from the VenusDB optimizer further reduce execution time by as much as 50%. It cannot be overlooked that many of these suggestions may not have been easily discovered by even the most experienced database administrators. Cycle time results demonstrate a similar performance improvement. This suggests that the duration of database locks by rule evaluation will also be reduced. Lastly, the programs that did not experience performance improvements via the VenusDB optimizer contained little data. Previous studies have demonstrated that VenusDB's LEAPS match algorithm successfully optimizes such programs [64].

In conclusion, the VenusDB optimizer facilitates practical development of hard active database applications by assisting rule developers in their optimizations tasks. In turn, the combination of VenusDB optimizer and VenusDB's LEAPS match algorithm provides scalable performance on both axes of hard active database programs: data and search.

TPC-D

Data Size	No Opt.	Push	Opt 1	Opt 2	Firings
1MB	1040	19	17	17	1079
10MB	11834	115	77	59	756
100MB	-	1130	494	488	727
1GB	-	16550	15451	11709	488
Least Sq	1199	17	16	12	-

TSP

Data Size	No Opt	Push	Opt 1	Opt 2	Firings
south	30	17	30	18	132
mountains	142	44	71	43	272
usa1	423	90	139	88	380
usa2	4801	847	684	674	1260
usa3	17945	3045	2319	2286	2550
usa4	-	6857	4990	4748	4032
Least Sq	1229	229	166	159	-

Waltz

Data Size	No Opt	Push	Opt 1	Opt 2	Firings
1	49	48	53	51	265
2	145	128	141	136	541
4	430	366	386	365	1093
8	1417	1124	1170	1139	2197
16	5519	3968	4056	3980	4405
32	-	17178	17518	17241	8821
Least Sq*	368	264	270	265	-

Manners

Data Size	No Opt	Push	Opt 1	Opt 2	Firings
16	7	7	8	6	184
32	37	23	23	23	624
64	345	110	102	97	12
128	4500	715	540	531	8640
Least Sq	42	7	5	5	-

REALESYS-A

Data Size	No Opt	Push	Opt 1	Opt 2	Firings
8	37	35	42	36	1098
16	67	68	78	68	1916
32	247	246	257	248	5597
64	523	517	531	519	10563
Least Sq	9	9	9	9	-

* Line estimation to datasize 16.

FIGURE 29. Complete Results

5.5 Acknowledgment

The empirical evaluation was performed at the Applied Research Laboratories at the University of Texas at Austin by John Williams. Williams' suggestions and patient work were invaluable to this study.

Chapter 6 Conclusion

In the last decade, active database research has received much attention. Many successful research prototypes have been demonstrated culminating with the extension of the SQL3 standard with a sophisticated trigger definition. However, applications of the technology have largely been restricted to *simple rule systems* where single rules encode entire programs. The development of more complex applications, termed *hard rule systems*, have been hindered by a host of limiting factors. These limitations include confusing and operational language semantics, flat monolithic-rule architectures, and limited performance. This dissertation addressed these obstacles in the following three ways.

First, a study of the semantics of the active database language VenusDB concluded that the software quality of rule programs using VenusDB can be improved by as much as 50% compared to rule programs using flat monolithic languages. VenusDB is an active database language derived from fixed point semantics. Additionally, VenusDB contains a formal definition for rule modules that conforms to the nested transaction model. Software metrics validate that these semantics improve overall software quality. Though not absolute, the improvements in metric ratings also suggest a reduction in maintenance costs for hard rule systems.

Second, the most general contribution of this dissertation addressed the complexity introduced by coupling modes. Coupling modes provide active database developers with a flexible mechanism for determining transaction semantics of rules. Unfortunately, they also introduce a level of complexity that is unmanageable in hard active database applications. Towards this end, this dissertation developed formal execution semantics and concurrency schemes for a subclass of hard rule systems called Log Monitoring Applications (LMAs). LMAs are expert

system applications that analyze logs maintained in a database. The concurrency proofs demonstrated that the write-once property of LMAs significantly reduce the number of applicable coupling modes. Specifically, the first set of proofs established that LMA^+ programs, LMAs with only positive variables, are confluent, and thus, rules can use the most flexible coupling modes. Since only the most flexible coupling modes are necessary, it is fair to conclude that a DBMS needs to only support these most flexible coupling modes in order to support the execution of LMA^+ programs. The second set of proofs established that for LMA^- programs, LMAs with both positive and negated variables, only one rule in a set of conflicting rules must be made atomic. However, the remaining rules may use more flexible coupling modes. An algorithm that builds upon the constructive nature of these proofs was presented to establish concurrency schemes for LMAs written in VenusDB. Together, the concurrency schemes and concurrency scheme assignment algorithm represent the first step in insulating application programmers from the details of integrating rules within a database.

Third, an optimizer is developed that assists database administrators in deploying scalable hard active database systems. Implemented within the VenusDB platform, the VenusDB optimizer uses component database statistics to optimize rules, decompose rule predicates into queries that are executed on component databases (termed predicate pushdown), and suggest a set of supporting indices. An empirical evaluation confirmed that this architecture successfully improves the scalability of VenusDB programs. The first set of results demonstrated that predicate pushdown often reduces the execution time of programs with large data sets by an order of magnitude. In these same programs, the optimizer's index suggestions further improve performance by as much as 50%. It cannot be overlooked that many of these index suggestions may not have been easily discovered by even the most experienced database administrators. The second set of

results demonstrated similar improvements in the reduction of cycle time, which in turn, reduces the duration of database locks due to rule evaluation. Since earlier empirical evaluations of VenusDB have demonstrated scalability of rule programs with little data and lots of search, it can be concluded that the addition of the VenusDB optimizer delivers scalability on both axes of hard active database programs: data and search.

In conclusion, the issues addressed in this dissertation represent significant steps towards facilitating hard active database applications.

6.1 Future Research

Although this dissertation addresses many issues that facilitate hard active database applications, other issues must still be addressed. These issues include, but are not limited to:

- This dissertation presented an investigation of LMA class of programs. However, many other useful application classes exist. In [29], a class of applications is presented that extends the LMA^- class by permitting deletes in rule actions. Still other classes vary with the complexity of the event algebra such as allowing composite algebras that may or may not include temporal and transactional events. Each of these classes of problems requires formal analysis. The resulting analysis would complete the general purpose rewrite system proposed in this dissertation. In addition to the techniques presented in Chapter 4, real-time analysis [68] and rule based optimization [41] techniques may be applicable.

- The constructive proof techniques of Chapter 4 utilize the results in confluent rule system, dependency graph, and serializability theories. These techniques are readily adapted for formal analysis. However, further parallelism may be gained through optimizing rule-based rewrites and dynamic analysis such as the

ones presented in [60,99]. Dynamic analysis allows for coupling mode assignments based on the semantic relationships that are only available at run-time (implying dynamic assignment of coupling modes). The net result may allow the restrictiveness of statically-stated coupling modes to be dynamically reduced to further increase system throughput.

- This dissertation does not address recovery issues. Most current active databases implement recovery using only their native extensional database system. This is insufficient. When an event is rolled back, all rules spawned from that event must also be rolled back. Further, any chained rules must be rolled back. Decoupled coupling modes further complicate matters [74]. For example, when a committed event spawns evaluation of a decoupled rule right before a system failure, the system must ensure that the decoupled rule is executed because its spawning event has already been committed. It is the author's opinion that the solutions to these problems may also be application dependent. For example, it may be sufficient for LMAs to implement recovery by using the nested transaction model combined with a persistent event log. Thus, chained rules will be rolled back and system failures would be logged so as to guarantee the execution of decoupled rules. Much work in the recovery of active rules has yet to be investigated.

- A limitation of the VenusDB modular definition is that it is procedural in the sense that rule modules cannot be inherited. Object-oriented rule systems have long adapted rule inheritance schemes within limited architectures [31]. Inheritance within VenusDB would be extremely powerful. Inheritance would have provided a method to implement differing Venus-based optimizer search strategies in a plug and play manner instead of using design templates as described in [92]. Search strategies could be elegantly exchanged at run-time to obtain significantly different results. Further, the design of REALESYS would have also

benefited from inheritance. In REALESYS, many modules contain “if(TRUE)” rules whose action consists solely of a module call that contains a single reused rule. Inheritance, especially if allowed to contain an overall priority assignment, would have eliminated many, if not all, of these “if(TRUE)” rules. It is the author’s conjecture that prioritized inheritance would be similarly beneficial to any application being developed in VenusDB.

- The VenusDB optimizer uses static optimization techniques. However, significant improvements in performance may be attained through mid-query re-optimizations such as the ones presented in [58]. Mid-query re-optimizations allow inefficient queries to be re-optimized during query execution. This is particularly important within a multidatabase system such as VenusDB. For example, a useful mid-query re-optimization includes interrupting execution when a rule is querying a web-based data source that is slow in responding. The optimizer can dynamically reformulate the rule in such a way that may have otherwise seemed inefficient.

In conclusion, this dissertation facilitates hard active database applications by solving many issues that limit their development. In turn, these solutions support the implementations of a new complexity of active database applications.

Appendix WatchDog: An LMA Application

This dissertation addressed issues facilitating hard active database applications. A driving force in the progression of this research was the development of WatchDog, a network security monitor that represents a "real world" example of an LMA [94][1]. Common to LMAs, the primary reason the developers of WatchDog chose a DBMS is to exploit the database's durability, integrity, and decision-support services for post event analysis. These characteristics allow near real-time monitoring of multiple network sensors, each simultaneously logging to a centralized database. When WatchDog discovers a network attack, the database's query facility readily supports further investigation by a network administrator.

This appendix reports on the architecture and performance of the WatchDog system. To date, WatchDog serves as the most comprehensive development effort within the VenusDB platform. In particular, WatchDog contains many rules (at the time of this writing, proposals for several hundred rules were being developed) and demanding data needs (~ 45 GB of data in the database at any one time). These demands required the exploitation of VenusDB's modular features (Chapter 3), the investigation of the LMA properties (Chapter 4), and the development of the VenusDB optimizer (Chapter 5). Therefore, the success of WatchDog speaks to the overall applicability of the active database contributions presented in this dissertation.

A.1 Overview

The monitoring for hacker attacks on some military sub-networks is performed by a centralized command. A privileged computer on each subnet, called a *probe*, sniffs all network traffic. An initial set of filters executing on that computer

1. WatchDog is currently being developed by the Rule-Based Expert System Project at the Applied Research Laboratories at The University of Texas at Austin.

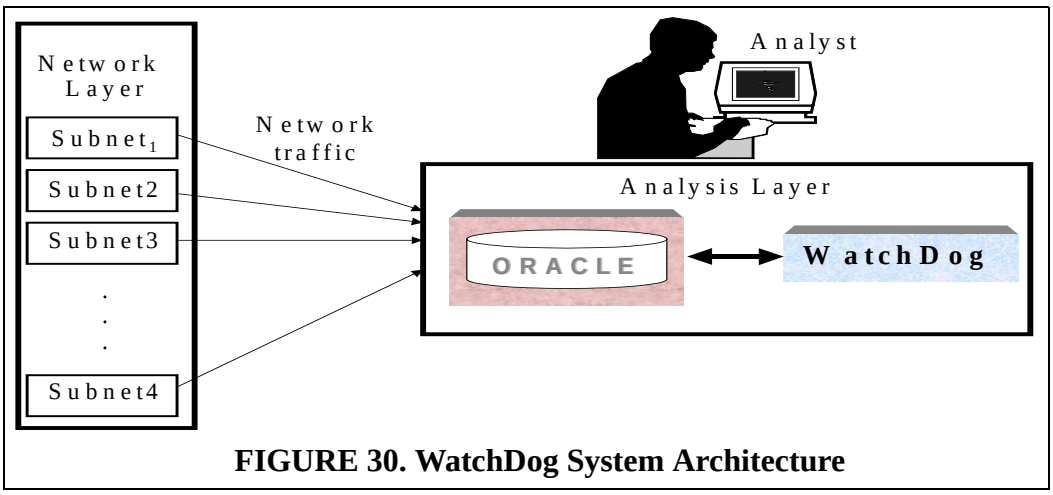


FIGURE 30. WatchDog System Architecture

analyzes traffic, captures suspicious packets while discarding obvious non-threatening connections. A synopsis of the results and a filtered subset of *connection logs* are forwarded to the central command and logged to an Oracle database. There, an ad-hoc C program with embedded SQL periodically wakes and further analyzes the activities. This analysis searches for possible intrusion patterns and raises alarms for the human network analysts if suspicious activity is detected. If further action is required, the network analyst may exploit an array of database query tools to investigate and track an individual alarm event.

WatchDog is an effort to better automate the analysis portion of this work process using the VenusDB active database system (Figure 30). In this respect, VenusDB is exploited to encode the possible intrusion detection component directly in declarative rules and to execute those rules in tight integration with the existing database in near real-time. Thus, the latency between the completion of a possible intrusion pattern and the notification of the human network analyst is significantly reduced.

WatchDog is also an example of an LMA - WatchDog receives connection logs and suspicious packets in near real-time and persists them within a database. These connection logs and suspicious packets can never be updated or deleted by

any application in order to enable computer forensics. Therefore, WatchDog's database is used a platform for durability and postmortem decision support. The LMA properties of WatchDog are exploited in the design of the rule architecture.

A.2 Implementation

WatchDog is implemented with only minimal modification to the existing database application.

A.2.1 Rule Architecture

Network traffic contains a high rate of activity. Further, hacker attacks must be caught as quickly as possible to reduce damage. Therefore, a requirement of WatchDog is to minimize the time to process an external event so as to maximize throughput. The techniques presented in Chapters 3.3 - 5 are exploited for this purpose.

First, the analysis techniques presented in Chapter 4 are used to specify nonrestrictive coupling modes. To facilitate system throughput, it was our initial goal to specify rules using decoupled coupling modes as often as possible. Further, WatchDog operates only on committed data. Therefore, E-C decoupled mode for all rules was adopted. However, our initial design called for C-A immediate mode for all rules. This is because VenusDB semantics define rules as atomic state transitions. Without formal analysis, we simply did not have the means to ensure proper behavior without specifying C-A immediate coupling mode. However, the completion of our formal investigation of LMA properties revealed that C-A decoupled mode was in fact sufficient to guarantee correctness within the current design. Further, we were able to provide a detailed list of requirements as to the rule properties that would ensure program correctness, or require more restrictive coupling modes. Developers have used these requirements to ensure maximum throughput.

Second, the rule architecture is modular. Since WatchDog is an LMA, rule processing necessarily begins with an insert into an extensional active LMA table. As an optimization, the event clause of all rules is limited to a single table[1]. Rules are segregated into modules based on this table. At run-time, insertion events invoke evaluation by passing data to the appropriate VenusDB module. The rules in the VenusDB modules are designed to seed their conditions with the data of their invoking event. This optimization tremendously reduces the condition search of individual rules. As described in our performance evaluation, this optimization combined with the VenusDB optimizer presented in Chapter 5 yield terrific performance.

A.2.2 Data Flow

Figure 31 illustrates the data flow of WatchDog. Data enters the analysis system through a bank of monitored networks and is inserted into an Oracle database. The Oracle connection log and suspicious packet tables are augmented with SQL triggers that fire upon insert events. These triggers alert and send copies of the inserted data to the appropriate VenusDB module by invoking methods of the VenusDB AMI [69].

Unfortunately, the implementation of this event detection mechanism is more complicated than it sounds. The fundamental constraint is the inflexibility of Oracle's trigger mechanism. This mechanism allows attaching a C function to the action of a trigger, which is executed within the Oracle process. VenusDB, however, is a C++ embedded system. It cannot coexist in the same process as Oracle. Therefore, VenusDB executes in a separate process and sends events from Oracle to VenusDB via AF_UNIX socket calls. More specifically, the Oracle registered C

1. The event clause for VenusDB rules is disjunctive. Therefore, this restriction does not restrict the semantic power of WatchDog rules.

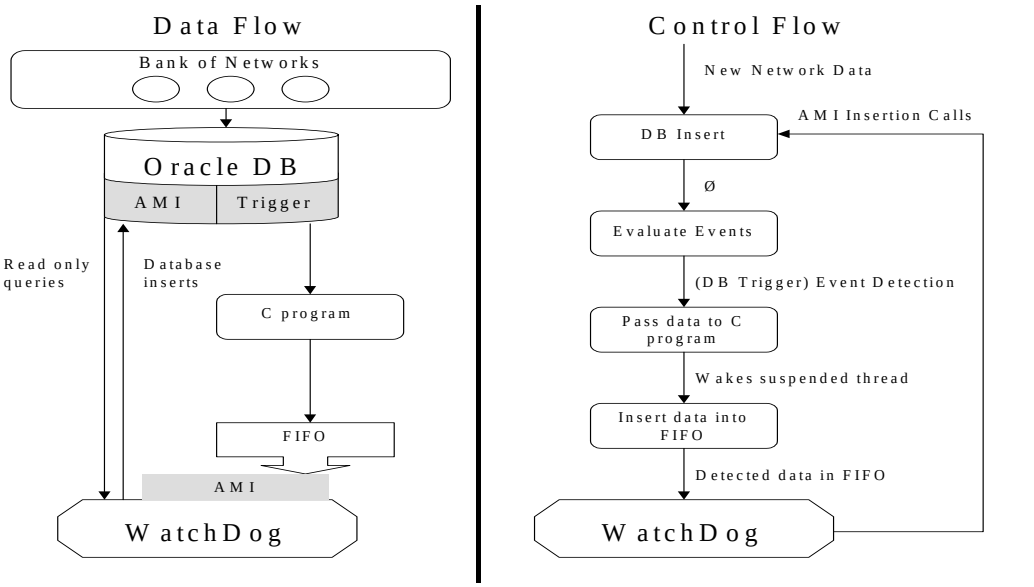


FIGURE 31. WatchDog Architecture

trigger function writes events to a socket, while a function in the VenusDB process monitors the socket for inserted events. When an event is detected, it is copied from the socket and placed into a FIFO queue. When VenusDB is ready to accept an event, it is removed from the queue.

The WatchDog rule component may then insert data from the Oracle database through the standard AMI methods.

A.2.3 Control Flow

Figure 31 also illustrates the control flow of WatchDog. As previously stated, data enters the system through a bank of monitored networks, causing the Oracle database to be updated. Oracle then evaluates which active LMA tables have been updated and passes control to the applicable trigger code. The trigger code then calls a C function that sends data to VenusDB via a socket interface. When the trigger code finishes, the database transaction completes. This architecture implements rules in E-C decoupled mode.

The WatchDog component contains a detached thread that monitors the socket for data and inserts detected data into a shared, semaphore-protected FIFO queue. Data is then removed from the queue and passed to the appropriate module at instances that are deemed safe. During rule evaluation, AMI calls may insert into the database starting the process over again. Note that since the database and the rule-set are in different address spaces, rule execution occurs in parallel with the database.

A.2.4 Code Modules

Four code units are added to the rule set to implement the system services.

- An Oracle AMI container and cursor implementation. This contains the code that 1) communicates with the Oracle database and 2) implements a detached thread that inserts data into an associated FIFO buffer.

- Oracle trigger code. This code passes data to the C function using a non-standard code escape.

- A C function that passes data through a socket call interface. This component must be written in a general-purpose programming language, not SQL. This is because it communicates with system sockets. The function is implemented as a shared object.

- A semaphore-protected FIFO data structure.

A.3 Measurements

The performance test were executed on a Sun Ultra-2 dual CPU machine with 256 megabytes of main memory and a 6-disk RAID Level 5 storage unit. The WatchDog application is instrumented to measure two interesting system properties.

First is an examination of the latency due to the database event detection scheme. This contains two components. The first is the latency of inserting a row into a table until the completion of its trigger action. This was measured at 20 milliseconds. It was constant regardless of the database size. Signaling by use of native event detection can be no faster than this. The second is the latency between inserting a tuple, writing the event information to the socket, and retrieving it from the socket, and notifying VenusDB through the AMI. This was measured at 23 milliseconds. Thus, the entire event detection scheme nominally adds 3 milliseconds. However, there are several distinct operating system processes involved in this scheme. Thus, there may be some amount of distributed execution, and similar results may not be achievable on a uniprocessor.

Second is a measurement of the program's *transaction time*, which is the time to execute the VenusDB portion per database insert[1]. The VenusDB portion is comprised of pulling the inserted tuple from the FIFO queue and executing the rules against the triggering tuple and database. The transaction time necessarily varies. This is because the amount of work done by the rule system depends on the added tuple and the state of the database. Evaluation may include chained rules, and each rule may require multiple database queries executed through the AMI. A data flow of 1,000 connection logs and 1,000 suspicious packet records was initiated to examine transaction time. A subset of these records contained values sufficient to cause rule firings. The initial state of the database at the beginning of the insertion flow is a test parameter. Results are shown in Table 8. In the first experiment, the database was empty. In the second, it contained 10,000 preexisting tuples. In the third, it contained 100,000 tuples. As shown by the uniformly low minimum transaction time, database inserts that do not trigger rule firings are

1. The transaction time is more precisely referred to as the time it takes for an external event to execute until quiescence, refer to Chapter 4. We use transaction time because of the associated implications of rule processing to database transactions.

	Empty DB	10,000 tuples	100,000 tuples	100,000 optimized
Average	.004	.04	.8	.3
Minimum	.0002	.0002	.0002	.0002
Maximum	.8	6	56	26
Std. Deviation	.04	.41	4.7	2.3

TABLE 8. WatchDog Metrics

quickly evaluated and discarded. This is true regardless of the size of the database, implying that certain events are eliminated based on trigger filtering only. The average and maximum transaction times scale roughly linear with database size. As a feasibility test, an early prototype of the VenusDB optimizer's schema suggestions was applied to the 100,000 tuple case. This optimization reduced execution times by approximately half.

The government security level of the Applied Research Laboratories' RBES project prohibits further analysis at this time.

A.4 Summary

This appendix reported on WatchDog, an effort to implement a complicated decision support application in an active database. The implementation moves functionality that was previously encoded outside an existing database and integrates it directly within the database. WatchDog is an LMA. As a result, LMA properties combined with VenusDB modularity were exploited for a clean and efficient rule architecture. By virtue of VenusDB's AMI, WatchDog utilizes the existing Oracle database. Performance was outstanding; on average it took less than one second to completely evaluate the impact of a database insert against a 100,000 tuple database. Performance was further improved two-fold with the addition of the VenusDB optimizer prototype. In conclusion, WatchDog demonstrates the effective application of the contributions of this dissertation.

References

1 A. Aiken, J. Hellerstein, and J. Widom, "Static analysis techniques for predicting the behavior of active database rules," *ACM Transactions on Database Systems*, vol. 20, no. 1, March, pp. 3-41, 1995.

2 A. Aiken, J. Widom, and J.M. Hellerstein, "Behavior of database production rules: termination, confluence, and observable determinism," in *Proceedings of the ACM-SIGMOD Conference*. San Diego, CA, June, 1992, 59-68.

3 H.J. Appelrath, H. Behrends, H. Jasper, and Olaf Zukunft, "Case studies on active database applications," *Database and Expert System Applications*. Zurich, Switzerland, September, 1996, pp.69-78.

4 J. Bachant, E. Soloway, and K. Jensen, "Assessing the maintainability of XCON-in-RIME: coping with the problems of a very large rule-base," In *Proceedings of the National Conference on Artificial Intelligence*. Seattle, WA, August, 1987, pp. 824--829.

5 J. Bailey, G. Dong, and K. Ramamohanarao, "Decidability and undecidability results for the termination problem of active database rules," *Symposium on Principles of Database Systems*. Seattle, WA, June, 1998, pp. 264-273.

6 C. Baral, J. Lobo, and G. Trajcevski, "Formal characterizations of active database: part II," *Deductive and Object Oriented Databases*. Montreux, Switzerland, December, 1997, pp. 247-264.

7 E. Baralis, S. Ceri, and S. Paraboschi, "Modularization techniques for active rules design," *ACM Transaction on Database Systems*, vol. 21, no. 1, March, pp. 1-29, 1996.

8 E. Baralis and J. Widom, "Using delta relations to optimize condition evaluation in active databases," *Rules in Database Systems*, Athens, Greece, September, 1995, pp. 292-308.

9 V. Barker and D. O'Conner, "Expert systems for configuration at digital: XCON and beyond," *Communications of the ACM*, March, 1989.

10 D. Batory. Class notes for CS 387H, *Database System Implementation*, University of Texas at Austin, Fall, 1996.

11 V. R. Basili, and R. W. Reiter, Jr., "Evaluating automatable measures of software development," In *Proceedings on Workshop on Quantitative Software Models*. October, 1979.

12 P.A. Bernstein and N. Goodman, "Concurrency control in distributed database systems," *ACM Computing Surveys*. vol 13, no. 2, June, pp. 185-221, 1981.

13 N. Bidoi and S. Maabout, "A model theoretic approach to update rule programs," *Proceedings of the International conference on Database Theory.* Delphi, Greece, January, pp.173-187, 1997.

14 D.G. Bobrow and M. Stefik. *The Loops Manual*. Xerox PARC, 1983.

15 A.J. Bonner, "Workflow, transactions and Datalog," *Symposium on Principles of Database Systems.* Philadelphia, PA, June, 1999, pp. 294-305.

16 D. Brant, "Inferencing on large data sets," Ph.D. dissertation. Austin, TX: Department of Computer Sciences, The University of Texas at Austin, 1993.

17 D. Brant and T. Grose, B. Lofaso, and D. P. Miranker, "Effects of database size on rule system performance: Five case studies," in *Proceedings of the 17th International Conference on Very Large Data Bases*. Barcelona, Spain, September, 1991, pp. 287-296.

18 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.

19 J.C. Browne, and et. al, "A new approach to modularity in rule-based programming," In *Proceedings of the 6th International Conference on Tools with Artificial Intelligence*, IEEE Press, 1994, 18-25.

20 L. Brownston, R. Farrell, E. Kant, and N. Martin, *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley Publishing Company, Inc., 1985.

21 A. Buchmann, J. Zimmermann, and J. Blakeley, "Building an integrated active OODBMS: Requirements, architecture, and design decisions," in *Proceedings of the 11th International Conference on Data Engineering*. Taipeh, Taiwan, March, 1995, pp. 117-128.

22 C. Bussler and S. Jablonski, "Implementing agent coordination for workflow management systems using active database systems," in *Proceedings of the*

4th International Workshop on Research Issues in Data Engineering. Houston, Texas, February, 1994, pp. 53-59.

23 M. Carey, D. DeWitt, J. Richardson, and I. Shekita, "Object and file management in the EXODUS extensible database system," in *Proceedings of the 12th International Conference on Very Large Databases*. Kyoto, Japan, August, 1986, pp. 91-100.

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

25 S. Chaudhuri and V. Narasayya, "AutoAdmin 'What-if' Index analysis utility," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Seattle, WA, June, 1998, pp. 367-378

26 S. Chaudhuri and V. Narasayya, "An efficient cost-driven index selection tool for Microsoft SQL Server," in *Proceedings of the 23rd Conference on Very Large Databases*. Athens, Greece, August, 1997, pp. 146-155.

27 M. Cherniack and S. B. Zdonik, "Rule languages and internal algebras for rule-based optimizers," *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Montreal, Canada, June, 1996, pp. 401-412.

28 Sunil Choenni, Henk M. Blanken, and D T. Chang, "On the selection of secondary indices in relational databases," *Date and Knowledge Engineering*, vol 11, no. 3, December, pp. 207-, 1993.

29 S. Comai and L. Tanca, "Using the properties of datalog to prove termination and confluence in active databases," *Rules in Database Systems.* Skövd, Sweden, June, 1997, pp. 100-117.

30 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, pp. 281-289.

31 J. Crawford, D. Dvorak, D. Litman, A. Mishra, and P. F. Patel-Schneider, "Path-based rules in object-oriented programming," in *Proceedings of the 13th National Conference on Artificial Intelligence*. Portland, Oregon, 1996, pp. 490-497.

32 D. Das and D. Batory, “Prairie: A rule specification framework for query optimizers,” in *Proceedings of the 11th International Conference on Data Engineering*, 201-210, Taipei, March, 1995, pp. 201-210.

33 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.

34 U. Dayal, M. Hsu, and R. Ladin, “Organizing long-running activities with triggers and transactions,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Atlantic City, NJ, 1990, pp. 204-214.

35 E. Dijkstra. *Formal Development of Programs and Proofs*, Addison-Wesley, 1990.

36 L. Do and P. Drew, “Active database management of global data integrity constraints in heterogeneous database environments,” in *Proceedings of the 11th International Conference on Data Engineering*. Taipei, Taiwan, March, 1995, pp. 99-108.

37 S. Flesca and S. Greco, “Declarative semantics for active rules,” *Database and Expert System Applications*. Vienna, Austria, August, 1998, pp. 871-880.

38 C.L. Forgy, “The OPS83 report,” Technical Report CMU-CS-84-133, Department of Computer Science, Carnegie Mellon University, May, 1984.

39 C. Forgy, “RETE: A fast match algorithm for the many pattern/many object pattern match problem,” *Artificial Intelligence*, vol., 19, pp. 17-37, 1982.

40 C. Forgy, “OPS5 user's manual,” Technical Report CMU-CS-81-135. Carnegie-Mellon University, 1981.

41 D. Gadbois and D.P. Miranker, “Discovering procedural execution of rule-based programs,” in *Proceedings of the 12h National Conference on Artificial Intelligence*. Seattle, Washington, July, 1994, pp. 459-464.

42 M. Gelfond and V. Lifschitz, "Representing action and change by logic programs," *Journal of Logic Programming*, vol. 17, no. 2,3, & 4, November, pp. 301-321, 1993.

43 M. Gelfond and V. Lifschitz, “The stable model semantics for logic programming,” *Proceedings of the 5th International Conference on Logic Programming*. St. Paul, MN, August, 1988.

44 A. Geppert, S. Gatziu, K. R. Dittrich, H. Fritschi, and A. Vaduva, “Architecture and implementation of the active object-oriented database management system SAMOS,” TR 95.29. Institut fur Informatik, Universitat Zurich, Switzerland, 1995.

45 J. Giarrantano. *CLIPS User’s Guide*. Artificial Intelligence Section, Lyndon B. Johnson Space Center, June, 1989.

46 J. Giarrantano and G. Riley, *Expert Systems: Principles and Programming, 3rd Edition*. Boston, MA: PWS Publishing Co, 1998.

47 G. Graefe and D. J. Dewitt, “The EXODUS optimizer generator,” in *Proceedings, 1987 ACM SIGMOD International Conference on Management of Data*. San Francisco, CA, May, 1987. 387-394.

48 G. Graefe and W. McKenna, “The Volcano optimizer generator: Extensibility and efficient search,” in *Proceeding of the 12th International Conference on Data Engineering*. Vienna, Austria, April, 1993, 209-218.

49 T. Grose, “The programming and functionality of OPS5 compared to LISP and FORTRAN in an aeronautical route planning system,” Master of Arts Thesis, The University of Texas at Austin, May, 1991.

50 A. Gupta, *Parallelism in Production Systems*, Pitman/Morgan-Kaufmann Publishers, Inc., Los Altos, CA, 1987.

51 H. Gupta, V. Harinarayan, A. Rajaramana, and J.D. Ullman, “Index selection for OLAP,” in *Proceedings of the International Conference on Data Engineering*. Birmingham, U.K., April, 1997, pp. 208-219.

52 E. Hanson, “The design and implementation of the Ariel active database rule system,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 8, no. 1, February, pp. 157-172, 1996.

53 S. Henry and D. Kafura, "Software structure metrics based on information flow," *IEEE Transactions on Software Engineering*, vol. 7, no. 5, September, pp. 510-518, 1981.

54 J.M. Hellerstein, “Predicate migration: Optimizing queries with expensive predicates,” in *Proceedings of the ACM-SIGMOD Conference on Management of Data*. Washington, D.C., May, 1993, pp. 267-276.

55 Inference Corp. *Art Reference Manual*, 1987.

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

57 N. Kabra and D. J. Dewitt, “OPT++: An object-oriented implementation for extensible database query optimization,” Unpublished paper http://www.cs.wisc.edu/shore/shore.papers.html, 1994.

58 N. Kabra and D. J. DeWitt, “Efficient mid-query re-optimization of sub-optimal query execution plans,” in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. Seattle, WA, June, 1998, pp. 106-117.

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

60 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.

61 T. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, December, 1976, pp. 308-320.

62 J. McDermott, “R1("XCON") at age 12: Lessons for an elementary school achiever,” *Artificial Intelligence*, vol. 59, 1993, 241-247.

63 D.P. Miranker, *TREAT: A new and efficient match algorithm for AI production systems*. Los Altos, CA: Pittman/Morgan-Kaufman Publishers, 1989.

64 D. P. Miranker, D. Brant, B. J. Lofaso, and D. Gadbois, “On the performance of lazy matching in production systems,” in *Proceedings of the 8th National Conference on Artificial Intelligence*. Boston, MA, July, 1990, pp. 685-692.

65 D. P. Miranker, F.H. Burke, J.J. Steele, J. Kolts, and D.R. Haug, “The C++ embeddable rule system,” *Int. Journal on Artificial Intelligence Tools*, vol. 2, no. 1, pp. 33-46, 1993. Also in the *Proceedings of the 1991 International Conference on Tools for Artificial Intelligence*

66 D. P. Miranker and L. Obermeyer, “An overview of the VenusDB active multidatabase system,” *International Symposium on Cooperative Database Systems for Advanced Applications*. Kyoto, Japan, December, 1996.

67 J. Misra. “UNITY: A foundation of parallel programming,” in *Proceedings. 9th International Summer School on Constructive Methods in Computer Science*. Marktoberdorf, Germany, July 24-August 5, 1988, in NATO ASI Series, Vol. F 55, ed. Manfred Broy, Springer-Verlag, pp. 397-433, 1989

68 A. Mok and R.H. Wang, “Response-time bounds of equal rule-based programs under rule priority structure,” *IEEE Transactions on Software Engineering*, vol. 21, no. 7, July, pp. 593-623, 1982.

69 L. Obermeyer, “Abstractions and algorithms for active multidatabases,” Doctoral Thesis. Austin, TX: Department of Compute Sciences, The University of Texas at Austin, 1999.

70 L. Obermeyer and D.P. Miranker, “Evaluating triggers using decision trees,” in *Proceedings of the 6th International Conference on Information and Knowledge Management*. Las Vegas, NV, November, 1997, pp. 144-150.

71 L. Obermeyer, L. Warshaw, and D. P. Miranker, “Porting an expert database application to an active database: An experience report,” *Databases: Active and Real Time. Baltimore*, MD, November, 1996.

72 M. Tamer Özsu, Adriana Munoz, and Duana Szafron, “An extensible query optimizer for an objectbase management system,” in *Proceedings of the 4th International Conference on Information and Knowledge Management*. Baltimore, MD, November 1995, 188-196.

73 F. Pachet, *Proceedings of the OOPSLA'94 Workshop on Embedded Object-Oriented Production Systems*. Technical Report LAFORIA 94/24. Laboratoire Formes et Intelligence Artifcielle, Institut Blaise Pascal. Dec., 1994.

74 N.W. Paton. *Active Rules in Database Systems*, Springer-Verlag New York, Inc., 1999.

75 A. J. Pasik, “A source-to-source transformation for increasing rule-based system parallelism,” *IEEE Trans. of Knowledge and Data Engineering*, vol. 4, no 4, August, pp. 336-343, 1992.

76 A. Pasik, “A methodology for programming production systems and its implications on parallelism,” Ph.D. Dissertation, Columbia University, New York, New York, 1989.

77 J. Pearl, *Heuristics. Intelligent Search strategies for Computer Problem Solving*. Addison-Wesley, 1984.

78 P. Picouet and V Vianu, "Expressiveness and complexity of active databases," *International conferences on Database Theory.* Delphi, Greece, January, 1997, pp. 155-172.

79 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, pp. 113-124.

80 H. Pirahesh, J. M. Hellerstein, and W. Hasan, "Extensible rule based query rewrite optimization in Starburst," in *Proceedings of the, 1992 ACM SIGMOD International Conference on Management of Data*. San Diego, CA, June, 1992, pp. 39-48.

81 L. Raschid, "Maintaining consistency in a stratified production system program," in *Proceedings of the 8th National Conference on Artificial Intelligence*. Boston, MA, August 1990, pp. 284-289.

82 E.Simon, J. Kiernan, and C. de Maindreville, "Implementing high level active rules on top of a relational dbms," in *Proceedings of International Conference on Very Large Databases*. Vancouver, British Columbia, Canada, August, 1992, pp. 315-326.

83 E. Soloway, J. Bachant, and K. Jensen, "Assessing the maintainability of XCON-in-RIME: Coping with the problems of a very large rule-base," in the *Proceedings of the National Conference on Artificial Intelligenc*e. Seattle, WA, July, 1987, pp. 824-829.

84 M. Staskauskas, "The formal specification and design of a distributed electronic funds-transfer system," *IEEE Transactions on Computers*, vol. 37, no. 12, December, pp. 1515-1528, 1988.

85 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.

86 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.

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

88 Sybase, Inc. Transact-sql user’s guide. Technical Report, 1987.

89 Transaction Processing Performance Council, “TPC benchmark D.” San Jose, CA, 1993.

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

91 Y-W Wang and E. N. Hanson, “A performance comparison of the Rete and TREAT algorithms for testing database rule conditions, “in *Proceedings of the Eighth International Conference on Data Engineering*. Tempe, AZ, February, 1992, pp. 88-97.

92 L. Warshaw and D.P. Miranker, “Rule-based query optimization, revisited,” *Proceedings of the 9th Conference on Information and Knowledge Management*. Kansas City, Kansas, November, 1999, pp. 267-275.

93 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.

94 L. Warshaw, L. Obermeyer, D. P. Miranker, and Sara Matzner, “VenusIDS: An active database component for an intrusion detection system,” unpublished document.

95 L. Warshaw and D. P. Miranker, “A case study of Venus and a declarative basis for rule modules,” in *Proceedings of the 5th Conference on Information and Knowledge Management.* Baltimore, MD, November, 1996, pp. 317-325.

96 D. Wells, J. Blakeley, and C. Thompson, “ Architecture of an open object-oriented database management system,” *Computer*, vol. 25, no. 10, 1992.

97 J. Widom, “The Starburst active database rule system,” *IEEE Transactions on Knowledge and Data Engineering*, August, 1996.

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

99 S-Y. Wu, D.P. Miranker, and J.C. Browne, “Toward semantic-based parallelism in production systems,” in *Proceedings of the International Conference on Parallel and Distributed Systems*, 1994.

100 C. Zaniolo, “The nonmonotonic semantics of active rules in deductive databases,” *Deductive and Object Oriented Databases*. Montreux, Switzerland, December, 1997, pp. 265-282.

101 C. Zaniolo, “Active database rules with transaction-conscious stable-model semantics,” *Deductive and Object Oriented Databases*. Signapore, December, 1995, pp. 55-72.

VITA

Lane Bradley Warshaw was born in Atlanta, Georgia on February 1, 1973, the son of Susie Blass Warshaw and Jerry David Warshaw. After completing his work at North Springs High School, Atlanta, Georgia, in 1991, he entered The University of Texas at Austin, where he graduated with special departmental honors with the degree of Bachelor of Science in Computer Sciences in 1996. Following graduation, he began working for the Applied Research Laboratories of The University of Texas at Austin, where he continued to work throughout his graduate studies. In August 1996, he entered the Graduate School of The University of Texas at Austin to pursue a Master of Science in Computer Sciences, which he was awarded in May 1999. In January 2000, he left the Applied Research Laboratories to begin working at Liaison Technology Incorporated, where he serves as a Senior Software Engineer.

Permanent address:1710 Goodrich Avenue #A, Austin, Texas 78704

This dissertation was typed by the author.