

An Overview of the VenusDB Active Multidatabase System

Daniel P. Miranker and Lance Obermeyer

Applied Research Laboratories
The University of Texas at Austin
Austin, TX 78712
miranker@cs.utexas.edu, lanceo@arlut.utexas.edu

Abstract

VenusDB is a C++ embedded, forward-chaining rule language and compiler that includes linguistic elements and runtime support for accessing multiple databases across multiple platforms. Multidatabase access was a natural evolutionary step for Venus. Evaluation of Venus using an expert-database application revealed the need for explicit syntax for the expression of event conditions. Thus, VenusDB provides for both event-condition-action (ECA) rules typical of active-database systems and condition action rules typical of expert systems and expert-database systems.

The Venus compiler is readily extended by virtue of an abstract interface, the AMI, that encapsulates the details of data access. Although middleware elements can be amorphous, the AMI forms a well defined interface for the encapsulation of databases and their integration with a forward-chaining inference engine.

Introduction

A forward-chaining rule system comprises a set of rules of the form $P_i \rightarrow A_i$, where P_i is a predicate on the state of the system and A_i is a procedure capable of changing that state and I/O. An inference engine executes a forward-chaining system by repeating a cycle where the predicates are evaluated, a subset of the satisfied rules is identified and their actions executed. There are a number of different definitions for each part of the system, the precise form of the predicate, the method for choosing a subset of satisfied rules and the termination conditions. Thus, as a programming paradigm there is considerable flexibility and applicability.

The integration of forward-chaining rule evaluation with databases to form *active databases* is proving to be valuable for the expression of core database functions, complex consistency constraints, workflow applications and computer aided manufacturing (CAM) applications[8,5]. Applications of these concepts are not limited to data in a single database at a single location. This is clearly the case for cooperative databases when it is important that changes in the values in one database be reflected by changes in values in another. For example, a just-in-time inventory management system might require a manufacturing request be recorded in a factory's database when an inventory amount in a warehouse's database drops below a specified amount. It should be clear from the example that this can be seen as maintaining a consistency constraint across two relations; however the constraint is complicated and the relations are located on different databases at different sites.

The integration of active properties with multiple databases may serve as a programming infrastructure that solves other problems in cooperative database including resolution of semantic inconsistencies [4].

Our work starts with the Venus language [3]. Venus is a forward-chaining language, embedded in C++, originally intended to support embedded hard-deadline real-time applications. Features of Venus include the ability to infer directly on data types defined in C++ (no translation to a lisp-like internal representation is required), implementation using the LEAPS match algorithm, and an underlying

execution model based on the nested-state model. The LEAPS algorithm can retain the operational semantics defined by the TREAT [10] and RETE [7] match algorithm and can also provide stronger performance guarantees than its predecessors due to reduced asymptotic algorithmic complexity. In particular, LEAPS requires only linear space vs exponential space for TREAT and RETE [11]. These properties are equally important for both embedded real-time systems and active and expert database applications.

The precise definition and implementation of VenusDB is the result of examining application needs in the database arena. The ALEXSYS mortgage pool allocation program, an expert-system now widely deployed in the financial securities industry is our most thoroughly exploited application [14,13]. The migration of these applications from batch-oriented, stand-alone programs to reactive multidatabase platforms demonstrated to us a clear need to distinguish some database updates as triggerable events and others as not. The rule language then had to be extended to include an explicit trigger condition. However, the traditional semantics of expert-systems requires that all events be trigger events. Therefore, the implementation was subtractive, not additive, and thus straight-forward.

Depending on how you count them, the authors have, over the years, developed 6 to 12 forward-chaining rule systems. Most of these systems have been developed as compilers, in the ordinary sense to the word. One consequence is that as the systems matured, encapsulating access to the underlying data became easier. Frankly, the motivation has been a clean definition of a target instruction set for rule-language compilers based on LEAPS and TREAT, similar to the Warren abstract machine (WAM) for Prolog [16] and the unnamed abstract instruction set developed for the RETE match. It is a fortunate coincidence that the abstract instruction set (AMI) for LEAPS allowed the current Venus compiler to be easily extended to use data from a variety of sources.

Language & Semantics

Venus

The original system in the Venus family is the main-memory expert system environment [3,12]. A Venus program is a collection of parameterized modules. Each module contains one or more rules. Modules may call each other non-recursively. The underlying semantics adopts a nested state model. The nested state model enables explicit, nonprocedural semantics for structured rule-based programming. A case study comparing Venus and OPS5 demonstrated dramatic improvements in four quantitative measures of code quality, suggesting that programmer productivity using Venus is much greater than in OPS and OPS-derived languages [17].

The parameters to a module may be both scalar values and collections of objects. The syntax for expressing rules is basically a legal C++ function of type `rule` containing a single `if` statement. An additional clause, plus overloaded definitions of “*” and “?” introduce quantification and inferencing over collections. See Figure 1.

The `from` clause, borrowed from SQL associates an alias with a cursor variable. Existentially quantified cursor variables are denoted by a “?”. Universally quantified variables are denoted by a “*”. Comparing to OPS5 derived languages, we define cursors rather than pattern or condition elements. It follows that it is more natural to use universal quantification (“for all”) instead of negation (“there does not exist”). The two approaches are formally equivalent.

The guard, (or `if` part) is a restricted C++ logical expression. It may refer to local variables, actual parameters, and function/method calls in the standard way. It may cause a search of the database of facts only by using existentially or universally quantified cursor variables.

For example, Figure 1 shows a module containing one rule that adds elements to a relation until the relation is symmetric. The parameter *r* is a container consisting of elements of externally defined C++ type *Relation*.

The Need for Events

Active Databases

Rules for active database systems comprise a three part form, known as *event-condition-action* (ECA). Depending on the system, the event and event qualifications may be simple or involve complex logical compositions. Table 1 enumerates the primitive events that systems' have included [9,8]. The intent is upon an event, the condition is evaluated. If the condition expression evaluation generates records, the records serve as parameters to an action.

Expert system languages have been limited to condition-action rules. Every update to the underlying state of the system is implicitly a trigger event for all rules. For example, if the condition mentions a table of employees, then any insert, update, or remove on that table triggers a reevaluation of the rule. It is the incremental nature of these updates and their interaction with the condition that yields clear requirements for specifying (filtering) which events may trigger rule evaluation. In a database environment where evaluating a condition and executing an action are significantly more costly (main memory speed vs. disk access speed), reducing the number of rule evaluations is imperative.

Figure 1 Venus Module & Rule

```
module enforce_symmetry(Relation r[])
{
    rule enforce;
    from r[?] x1; // existential
    from r[*] x2; // universal
    if(x1.domain != x2.range ||
        x1.range != x2.domain) {
        Relation tmp;
        tmp.domain = x1.range;
        tmp.range = x1.domain;
        r.insert(tmp);
    }
}
```

Table 1 Standard Events

Event Type	Events
Database Event	insert, update, remove - relational db method call - object-oriented db
Transaction Event	begin, commit, abort
Temporal Event	absolute, relative, periodic
User Defined Event	application specific

Application Requirements

Our analysis of application requirements is based largely on our experience with the ALEXSYS program [14]. That program is from the domain of finance. It takes items from inventory (collections of individual mortgages with similar characteristics called *pools*) and matches them with orders (orders to buy mortgages called *contracts*) according to a variety of government regulations. Pools can be split into smaller items, much as a log can be cut into smaller pieces. The ALEXSYS program encodes a set of prioritized heuristics that dictate how to combine pools to satisfy a contract and when to split large pools into groups of smaller pools. The program proceeds by continued greedy application of the heuristics to fill as many contracts as possible. Unfillable contracts are possible but not common. New contracts and new pools are continually received.

The original program was developed as a batch-oriented expert system program. It was run on all data, once at the end of the month. We felt that the continuous receipt of contracts and pools made this program an excellent candidate for implementation on an active database. Therefore, we ported the program to VenusDB [13].

Our port of ALEXSYS to VenusDB demonstrated a clear requirement for event filtering in an active database system. The rules in ALEXSYS were roughly segregated into rules responding to new contracts and rules responding to new pools. The modular structure of the VenusDB language allowed the two

groups to be segregated into two modules. A top level rule with the event statement set to pool sensitivity guards the new pool module. Similarly, a top level rule with the event statement set to contract sensitivity guards the new contract module. Thus, though the number of rules that actually have stated event descriptors is small, nearly all rules in the system “inherit” an event specification from a higher level module. This segregation of the rulebase into active and inactive groups based on the event was a critical part of the implementation.

Figure 2 VenusDB Module & Rule

```
module enforce_symmetry(Relation r[]) {
  rule enforce;
  from r[?] x1;      // existential
  from r[*] x2;      // universal
  event x1 insert;   // sensitive
  event x2 none;     // insensitive
  if(x1.domain != x2.range ||
     x1.range != x2.domain) {
    Relation tmp;
    tmp.domain = x1.range;
    tmp.range = x1.domain;
    r.insert(tmp);
  }
}
```

VenusDB

VenusDB attempts to unify active database systems with expert database systems by supporting event specification through ECA rules and supporting Venus (OPS5 [6] derived) execution semantics.

Event Specification

VenusDB implements its ECA language by adding an event specification to the standard Venus language. The only supported events are the database events **insert**, **update**, and **remove**. Existentially quantified variables are sensitive only to **insert** and **update**. Universally quantified variables are sensitive only to **update** and **remove**. This is because a rule that was not satisfied due to a failure in an universally quantified variable can only become satisfied if the blocking element is either changed or removed. Primitive variables are sensitive only to **update**. Temporal events are not directly supported, though coarse grained temporal events are simulated by explicitly calling a VenusDB module (e.g. calling VenusDB from a **cron** entry).

In VenusDB, an event of **all** is the default behavior and is equivalent to standard (condition-action) Venus. Specifying events either has no effect on the rule (event **all**) or reduces the number of rule firings by making some rules insensitive to database updates that they otherwise would have been sensitive to.

Example

Figure 2 shows the rule from fFigure 1 in VenusDB syntax. The only difference is the addition of **event** statements for the two cursor variables. Cursor **x1** is existentially quantified over the relation **r**, and is sensitive only to the **insert** event. This implies that VenusDB will trigger the rule on the initial contents of the container and on inserts to the container. Event **update** is not listed because it will never happen within this module. Cursor **x2** is explicitly set to event **none**. This is because a universally quantified cursor is only sensitive to **update** or **remove**. Since neither of these happen within this module, some of the overhead of monitoring for events is avoided.

VenusDB Architecture

The VenusDB system is a compiled system. Rule processing is managed by an implementation of the LEAPS algorithm. Access to databases is handled using the Venus Abstract Machine Interface (AMI).

A VenusDB application is implemented in three steps.. First, rule source code is written. Second, the rule source is processed by the VenusDB compiler, which generates a combination of C++ code and

instructions from the AMI. Finally, the C++ code, AMI instructions, AMI implementation library (libraries), and user source code and libraries are compiled by the user's C++ compiler into an executable application.

The compiler is organized conventionally. It includes a lexer/parser that translates rules into intermediate forms similar to query graphs. Various optimization phases operate on the intermediate forms. C++ code is generated, primarily by traversing the forms.

Optional inputs to the optimization phase of the compiler include schema information consisting of the physical capabilities of the databases (especially available indices) and query plans instructing the compiler to output specific rule execution plans for specified rules [12].

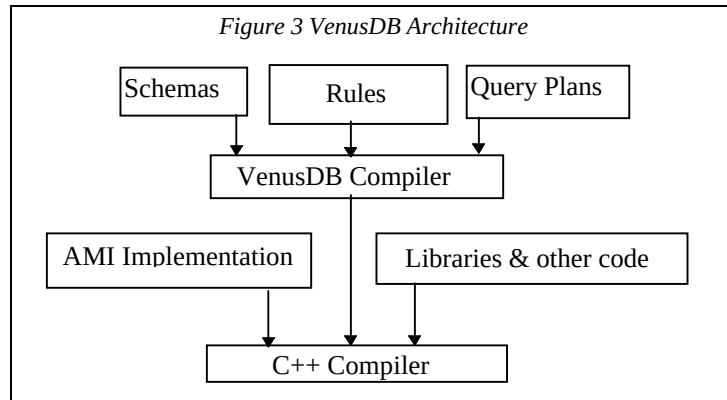
Abstract Machine Interface

Venus is based on a demand-driven matching algorithm. Until recently, the best (and still most popular) match algorithms have been discrimination networks like TREAT[10] and RETE [7]. These, and virtually all other match algorithms that support the execution of forward-chaining systems, have asymptotic algorithmic space complexity that grows exponentially in the size of the rules. Although worst-case is rarely achieved, the memory resources used by those algorithms are volatile and do not scale to databases [11,2,15]. The LEAPS algorithm, which is used by Venus and VenusDB, executes its steps in a demand driven fashion, similar to the lazy evaluation of functional programs, such that it requires minimal internal state. The asymptotic algorithmic space complexity of LEAPS is linear in size of the database. Over a suite of test programs demonstrates only trivial memory requirements beyond storing the database.

The problem is that eager match algorithms enumerate all instantiations of all satisfied rules on each cycle. The memory required to store that enumeration, alone, is exponential in the number of clauses in the rule. The memory is consumed even though, in sequential executions, only one instantiation of one rule is fired per cycle. The LEAPS algorithm avoids the enumeration by using the heuristics used to identify the winning rule as a means of sorting the search space. LEAPS then conducts a best-first search insuring that the first rule instantiation that it computes is the same rule instantiation that wins by applying the heuristics as a sort predicate on the enumeration of rule instantiations.

It has been our position, as well as others, that the success of active-databases is tied to the effectiveness of the implementations, and that TREAT and RETE, despite their often transparent mapping to database implementations are incapable, algorithmically to support this application area [11,15,10]. Yet, initial understanding of LEAPS does not reveal an obvious method for integrating its execution with a database. A special challenge behind this research has been the determination of an appropriate abstraction for the interface to the data sources.

The clearest algorithmic description of the interface can be found in Batory and Thomas [1]. The key elements of the abstraction involved identifying a cursor-based definition of LEAPS, thus data could be streamed into the inference engine. Elements of the internal state and sequencing in LEAPS has been



mapped to precise cursor semantics. The AMI improves upon the work of Batory and Thomas by further abstracting data access.

Table 2 Passive Container

Function	Description
<code>newCursor()</code>	create a new cursor over the container

VenusDB executes by issuing commands from the abstract machine interface, an instruction set used by LEAPS programs. The AMI is modeled after the Warren abstract machine, which defines the instruction set used by Prolog programs [16]. The purpose of the AMI is to encapsulate all database functionality behind a common interface.

The AMI is defined by a set of abstract C++ classes. An implementation of the AMI for a particular database is two C++ classes that inherit from the appropriate AMI base class, a container implementation class and a cursor implementation class. AMI implementation classes use the C++ `template` facility to provide type safety.

Data Model

The AMI uses the C++ data model. Databases that do not natively use the C++ data model (e.g. most relational systems) require translation into C++ forms. For example, a relation R with integer attributes a and b might be represented in C++ as a class `Rclass` with public `int` data members a and b . Translation into C++ is done by the AMI implementation. Recall that VenusDB is embedded into C++.

Tuple ID

The AMI assumes each tuple/object has an identifier. The identifier must identify both an object **and** a version of the object. Each time an object is modified, it must be given a new identifier. VenusDB uses the identifier to implement a form of tuple version detection.

Containers

Containers encapsulate database storage mechanisms. There are two fundamental container types, passive and active. Passive containers are read-only, while active containers are read-write. This distinction exists because a great many database tables are static¹, such as a table of English to metric conversion values.

The passive container support the `getCursor()` function, which instantiates and returns a passive cursor (discussed next).

Active containers support the `getCursor()` function, which instantiates and returns an active cursor (discussed next). Active containers also support the function `insert()`. This function is called by VenusDB only. Other applications do not have to use this function; they are free to use the normal database insert functions.

Table 3 Active Container

Recall that in VenusDB, data containers used in modules are either local to the module or passed into the module as parameters.	Function	Description
	<code>newCursor()</code>	create a new cursor over the container
	<code>insert(T)</code>	insert a tuple T into the container

¹ Static is defined with respect to an execution of the rule application. A table of United States Presidents is not truly static, but it changes at such a slow rate as to be not worth considering.

Local data elements are by definition temporary, and are allocated from transient memory. Parameters to root modules are persistent with respect to VenusDB. An AMI container instance defined outside of VenusDB takes a persistent database table as a construction argument. Thus an AMI container can serve as a wrapper around an existing database table.

Cursors

Cursors encapsulate iteration over database elements stored in containers. There are two fundamental cursor types, passive and active. Passive cursors iterate over passive (read-only) containers. Active cursors iterate over active (read-write) containers.

Both active and passive cursors share much common functionality. The `fetch()` function searches the container for a particular element and positions the cursor to the element. The execution of LEAPS results in sporadic execution of searches. Cursors being iterated will often be suspended for long periods, and resumed at some later point. The beginning of a suspension will be signaled by the `suspend()` function, and completion by the `resume()` function. During the time a cursor is suspended, the rule system continues to fire rules. These rule cannot, by definition, modify elements in passive containers. However, they may modify elements in active containers. This implies that a suspended cursor **must not maintain a lock** on either the table or the suspended element.

Table 4 Cursors

Function	Description
<code>getData()</code>	Get the data of the current element.
<code>getId()</code>	Get the tuple ID of the current element.
<code>fetch(ID)</code>	Position to an element. ID is a previous return value from <code>getId()</code> .
<code>suspend()</code>	Stop iteration
<code>resume()</code>	Resume iteration. Return <code>TRUE</code> if element suspended on is valid, <code>FALSE</code> if element has been deleted.
<code>reset()</code>	Reposition cursor to beginning of extent
<code>next()</code>	Position cursor to next element in extent

A cursor iterates over a set of elements in its container, which we term an *extent*. The extent of a passive cursor is all elements inside its container.

An active cursor has the additional feature that it supports modifying the element at the current position and deleting the element at the current position.

The main complexity when implementing the AMI cursor classes is in defining the extent of an active cursor. At the point when an active cursor is created, all elements in the container are in the extent. As time progresses, elements removed from the container are removed from the extent.

Table 5 Active Cursors

Function	Description
<code>remove()</code>	Remove the current element.
<code>modify(T)</code>	Update the current element. Replace with the value of T.

Elements added to the container are **not** added to the extent. Thus, an active cursor can be viewed as **sensitive to removals and insensitive to insertions**. If the element at the location where an active cursor has been paused using `suspend()` has been deleted, the `resume()` function positions the cursor to the next element in the extent.

Event Signaling

The database is responsible for signaling the rule system whenever its state has changed. Depending upon the application, this signaling may be on committed or uncommitted data. In databases that support triggering, this may be entered as a trigger. In databases that do not support triggers, some form of application specific inline wrapper function is necessary.

Signaling may be on committed or uncommitted data, depending upon the application. If committed data is signaled, then VenusDB will, by necessity, operate in a detached coupling mode [9]. If uncommitted data is signaled, then VenusDB will operate in either immediate or deferred coupling mode. If this is the case, then the rule system must be activated and allowed to run before a commit is issued.

Table 6 Event Signaling

Function	Description
<code>SignalEvent(EVENT, data, id)</code>	Signal a database event to VenusDB .

The `SignalEvent()` function is public method of a base class of the AMI container class. Thus each container instance has a `SignalEvent()` function. The `event` parameter is one of `INSERT` or `REMOVE`. Update is handled as a remove followed by an insert. The `data` parameter is the data of the effected element. The data is required for certain indexing and optimization steps. Note that the data value is not copied. It is used only during the duration of the `SignalEvent()` function call. The `id` parameter is the object identifier as discussed above. The `id` will be maintained for an arbitrarily long period. Its only use will be as a parameter to a cursor `fetch()` function.

Universal Quantification

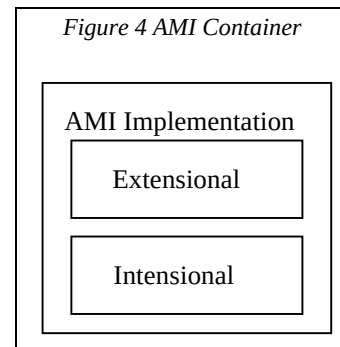
The VenusDB system supports universal quantification (OPS5 negation). An unfortunate result of the current implementation of the LEAPS algorithm is that certain pathologic cases can cause a rule that includes a universally quantified cursor variable to fire multiple times on a single unique combination of data elements. This violates the “once” semantics of VenusDB. The LEAPS solution to this is to maintain *shadow* elements, which are special data elements that block subsequent firings of a rule on a specific combination of data elements [11].

A shadow element may need to be created whenever an element is removed from a container that has an active universally quantified cursor variable. An existing shadow element may safely be removed when there are no active universally quantified cursor variables over the container. These conditions are not apparent to the user, but they are apparent to the VenusDB runtime implementation. Therefore, **the AMI completely hides this complexity from users**.

The AMI segregates a data container into *extensional* and *intensional* areas². The extensional area is the data area defined by the database table. The intensional area is a private to VenusDB area that contains shadow elements. Management of the intensional area is done by VenusDB. In fact, the user has no idea that the intensional area even exists.

Current Status

VenusDB is implemented, and several test applications have been created and demonstrated. The AMI has been implemented over several data sources, and programs have been instantiated with multiple AMI implementations. VenusDB and the AMI have both matured to the point where graduate students have been able to create applications and implement the AMI.



Programs

Several test programs have been executed. Tests have been executed using the different AMI implementations **and** mixing the different AMI implementations.

The most substantial program executed to date is the ALEXSYS mortgage pool allocation program. This program consisted of 102 rules. Different AMI implementations were used in the program, including multiple AMI implementations at the same time (multidatabase access).

AMI Implementations

The AMI has been implemented over a variety of storage types. All implementations have been successful. Where possible, existing libraries were used to implement as much as possible. A future paper will report a full performance characterization of each implementation.

- Rogue Wave containers - main memory double linked list from commercial class library.
- ObjectStore - commercial object-oriented database system.
- Sybase - commercial relational database.
- HTTP - iterate over http tables (passive mode only).

Conclusions

This paper presented an overview of VenusDB, an active multidatabase. VenusDB supports active behavior over heterogeneous databases. Active behavior is encoded in event-condition-action rules, which VenusDB supports by adding event qualifications to the condition-action rules of Venus. The first contribution of the paper is a discussion of our experience porting an expert database program to VenusDB, and the changes to the VenusDB language that we were forced to make. The second contribution is a presentation of the Venus abstract machine interface (AMI). The AMI hides LEAPS specific data access behaviors behind a simple interface. A user of VenusDB that wishes to inference over data stored in databases need only implement the AMI on the target database. By virtue of the AMI, multiple databases may be seamlessly accessed in a program.

Future work in the Venus program remains. Promising areas of research include distributing more database work down to component databases, inferencing over unusual data sources, especially World

² Borrowing the terminology of Datalog.

Wide Web sources, transaction management, and concurrency improvement. Research into these issues, as well as application development, continues at the Applied Research Laboratories, where we have the luxury of a steady stream of interesting applications to guide our work.

References

1. Batory, D, J. Thomas, and M. Sirkin. "Reengineering a Complex Application Using a Scaleable Data Structure Compiler." *In Proceedings of the 1994 ACM SIGSOFT Conference*, 1994.
2. Brant, D., 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 Databases*, 1991.
3. Browne, J. C., and et.al. "Modularity in Rule-Based Programming." *International Journal on Artificial Intelligence Tools*, 4(1), 1995.
4. Ceri, S., and J. Widom, "Managing Semantic Heterogeneity with Production Rules and Persistent Queues." *In Proceedings of the 19th International Conference on Very Large Databases*, 1993.
5. Chakravarthy, S. "A Comparative Evaluation of Active Relational Databases," UF-CIS-TR-93-002. Computer and Information Sciences, University of Florida, 1993.
6. Forgy, C. "OPS5 User's Manual." Technical Report CMU-CS-81-135. Carnegie-Mellon University, 1981.
7. Forgy, C., "RETE: A Fast Match Algorithm for the Many Pattern/Many Object Pattern Match Problem. *Artificial Intelligence*, 19, 1982.
8. Hanson, E. and J. Widom, "An Overview of Production Rules in Database Systems." *The Knowledge Engineering Review*, 8(2), 1993.
9. McCarthy, D. R. and U. Dayal. "The Architecture of an Active Database Management System." *In Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, 1989.
10. Miranker, D. P., and B. Lofaso. "The Organization and Performance of a TREAT-Based Production System Compiler." *IEEE Transactions on Knowledge and Data Engineering*, 3(1), 1991.
11. Miranker, D. P., D. Brant, B. J. Lofaso, and D. Gadbois. "On the Performance of Lazy Matching in Production Systems." *In Proceedings of the 1990 National Conference on Artificial Intelligence*, 1990.
12. Obermeyer, L., D. P. Miranker, and D. Brant. "Selective Indexing Speeds Production Systems." *In Proceedings of the 7th International Conference on Tools with Artificial Intelligence*, 1995.
13. Obermeyer, L., L. Warshaw, and D. P. Miranker. "Porting an Expert Database Application to an Active Database: An Experience Report." Submitted for publication, 1996.
14. Stolfo, S. and et.al. "The ALEXSYS Mortgage Pool Allocation Expert System: A Case Study of Speeding Up Rule-based Programs." Columbia University Department of Computer Sciences and Center for Advanced Technology, 1990.

15. Stonebraker; M. "The Integration of Rule Systems and Database Systems." *IEEE Transactions on Knowledge and Data Engineering*, 4(5), 1992.
16. Warren, D. H. D. "An Abstract Prolog Instruction Set." Technical Note 309. Artificial Intelligence Center, SRI International, 1993.
17. Warshaw, L. and D. P. Miranker. "A Case Study of Venus and a Declarative Bases for Rule Modules." " In *Proceedings fo the 5th conference on Information and Knowledge Management*, 1996