Active Knowledge Based Systems:

Techniques and Applications

N. Bassiliades and I. Vlahavas

Dept. of Informatics,

Aristotle University of Thessaloniki,

54006 Thessaloniki,

Greece

Tel: +30-31-998433 **Fax:** +30-31-998419

E-mail: {nbassili,vlahavas}@csd.auth.gr

URL: http://www.csd.auth.gr/~plk

Abstract

This chapter focuses on Active Knowledge Base Systems and more specifically it presents various implementation techniques that are used by the numerous systems found in the literature and on applications made based on such systems. Systems are compared based on the different techniques and on their efficiency on various applications. Finally, the Active Object-Oriented Knowledge Base System DEVICE is thoroughly described giving emphasis on its advantages against similar systems. Furthermore, two applications based on the DEVICE system are described: Deductive Databases and Data Warehouses.

Table of Contents

I. INTRODUCTION

II. ACTIVE DATABASE AND KNOWLEDGE BASE SYSTEMS

A. THE RULE SPECTRUM
B. ACTIVE DATABASE SYSTEMS
1. ECA Rules
2. Production Rules
C. ACTIVE KNOWLEDGE BASE SYSTEMS
1. Integration of Events in Production Rule Conditions
2. Integration of Production Rules in Active Databases

III. DEVICE: AN ACTIVE OBJECT-ORIENTED KNOWLEDGE BASE SYSTEM

A. SYSTEM ARCHITECTURE
B. THE PRODUCTION RULE LANGUAGE
C. INTEGRATION OF PRODUCTION RULES
1. Compilation and Matching of Rule Conditions

IV. APPLICATIONS OF ACTIVE KNOWLEDGE BASE SYSTEMS

A. DEDUCTIVE DATABASES
1. Common Semantics for Production and Deductive Rules
2. Implementation of Deductive Rules in DEVICE
B. DATA WAREHOUSING
1. System Architecture
2. View Materialization and Maintenance
3. Integration of Heterogeneous Data

V. CONCLUSIONS AND FUTURE DIRECTIONS

APPENDIX

REFERENCES

I. Introduction

Knowledge is the information about a specific domain needed by a computer program in order to exhibit an intelligent behavior over a specific problem. Knowledge includes both information about real-world entities and relationships between them. Furthermore, knowledge can also take the form of procedures on how to combine and operate on the above information. Computer programs that encapsulate such knowledge are called *knowledge-based systems*.

Knowledge is usually captured in some form of human logic and programmed through non-deterministic, declarative programming languages, such as Prolog and OPS5. These languages allow the programmer to define in a highly descriptive manner the knowledge of a human expert about problems and their solutions. Furthermore, programs written in such languages can be extended easily because the data and program structures are more flexible and dynamic than the usual.

Contemporary real-world computer applications try to model the complex and vast amount of the modern society's knowledge that must be handled by knowledge-based systems. More "traditional" applications suffer similarly from the existence of large amounts of data, which are equivalent to facts in the context of knowledge-based systems. The traditional solution is to couple the programs that process data with special systems devoted to the efficient and reliable storage, retrieval and handling of data, widely known as Database Management Systems (DBMSs).

The same trend is followed for knowledge-based systems where the management of knowledge has moved from the application to the Knowledge Base Management Systems (KBMS). KBMSs are an integration of conventional DBMSs with Artificial Intelligence techniques. KBMSs provide inference capabilities to the DBMS by allowing the encapsulation of the knowledge of the application domain within the database system. Furthermore, KBMSs provide sharing, ease of maintenance, and reusability of knowledge, which is usually expressed in the form of high-level declarative rules, such as production

and deductive rules. The Knowledge Base System (KBS) consists of the KBMS along with a specific set of rules (called the *rule base*) and data or facts (called the database). The rule base and the database of a KBS are collectively called the Knowledge Base (KB).

A recent trend to bridge the gap between knowledge base and database systems is active database systems. Active database systems constantly monitor system and user activities. When an interesting event happens they respond by executing certain procedures either related to the database or the environment. In this way the system is not a passive collection of data but also encapsulates management and data processing knowledge.

This re-active behavior is achieved through active rules which are a more low-level, procedural counterpart of the declarative rules used in knowledge-based systems. Active rules can be considered as primitive forms of knowledge encapsulated within the database; therefore, an active database system can be considered as some kind of KBS.

All rule paradigms are useful for different tasks in the knowledge base system. Therefore, the integration of multiple rule types in the same system is important. This will provide a single, flexible, multi-purpose knowledge base management system where users/programmers are allowed to choose the most appropriate format to express the application knowledge.

The objective of this chapter is to discuss some of the existing approaches to building a KBMS by integrating one or more rule types in a DBMS, giving emphasis on solutions based on the re-active behavior of Active Knowledge Base Systems. The implementation techniques found in various literature systems are presented and compared according to their functionality and efficiency.

Finally, the chapter presents in detail the Active Object-Oriented Knowledge Base System DEVICE that integrates multiple rules types into an active OODB system. Furthermore, applications based on the DEVICE system, such as Deductive Databases and Data Warehousing are discussed.

The rest of this chapter is as follows; Section 2 presents active databases and the various approaches followed towards Active Knowledge Base Systems by unifying multiple rule types into a

single system. In section 3, we present in detail the DEVICE system, which integrates data-driven (production) rules into an active OODB system that supports generically only event-driven (ECA) rules. This section presents the rule language, the compilation scheme, the rule matching algorithms, and the rule semantics of the DEVICE system. Section 4, presents two applications based on the DEVICE system, namely Deductive Databases by supporting deductive rules and derived data through production rules and Data Warehousing by coupling DEVICE with a multi-database system. Finally, section 5 concludes this chapter with a discussion of current and future directions for active knowledge base systems.

II. Active Database and Knowledge Base Systems

In this section we overview active database and knowledge base systems. Specifically, we present various techniques for implementing active rules into a database system and on unifying high-level rules into an active database system, resulting in an active knowledge base system.

A. The Rule Spectrum

Knowledge Base Management Systems (KBMSs) are normal Database Management Systems (DBMSs) extended with some kind of knowledge. *Knowledge* usually means some kind of declarative language, and takes the form of rules [1]. According to which rule type has been integrated into a DBMS, we distinguish between two types of KBMS: deductive and active database systems.

Deductive databases [2, 3, 1] use declarative logic programming style rules (also called *deductive rules*) that add the power of recursively defined views to conventional databases. Deductive rules describe in a declarative manner new, derived data in terms of existing data, without an exact description of how new data are created or treated.

On the other hand, active database systems extend traditional database systems with the ability to perform certain operations automatically in response to certain situations that occur in the database. For

this reason they use low-level *situation-action* rules (also called *active rules*) which are triggered when a *situation* arises in the database. As a consequent a set of *actions* is performed on the database. Active rules can be used to provide for varying functionality to the database system, such as database integrity constraints, views and derived data, authorization, statistics gathering, monitoring and alerting, knowledge bases and expert systems, workflow management, etc. Active rules can take the form of *data-driven* or *event-driven* rules.

Data-driven or *production* rules are more declarative than event-driven rules [4] because their *situation* part is a declarative description of a firing situation (a query) without an exact definition of how or when this situation is detected.

Event-driven or Event-Condition-Action (*ECA*) rules are more procedural because they explicitly define their triggering situation [5]. Specifically, ECA rules are triggered by an *event* that occurs inside or outside the system, then a *condition* is checked to verify the triggering context and, finally, the *actions* are executed.

Despite the differences of the above rule types in their syntax, semantics, use, and implementation, it has been proposed by Widom [5] that active and deductive rules are not distinct but rather form a spectrum of various rule paradigms. Widom has described a general common framework under which all rule types found in the literature can be placed by adapting slightly the framework. **Figure 1** shows how the above rule types fit into the rule spectrum.

All rule paradigms are useful in an active KBMS. Therefore, the unification of the various rule types in a single system is an important research task that has received considerable attention over the recent literature.

According to Widom, higher-level rule can be translated into (and, therefore, emulated by) lower-level rules. Furthermore, the semantics of high-level rules can be extended to cover the semantics of lower-level rules, so that the latter can be used in a system that supports a high-level rule system.

In the following subsections of this section, we present in more detail various approaches to active databases, and then, we discuss various techniques to unify some or the entire set of the above rule types. Notice that the presentation of implementing deductive rules over an active database is deferred until section IV, where deductive databases is described as an application of active knowledge base systems.

B. Active Database Systems

An *Active Database System* (ADB) is a conventional, passive database system extended with the capability of reactive behavior. This means that the system can perform certain operations automatically, in response to certain situations that have occurred in the database.

An ADB is significantly more powerful than its passive counterpart because it can achieve the following:

- perform functions that in passive databases systems must be encoded in applications;

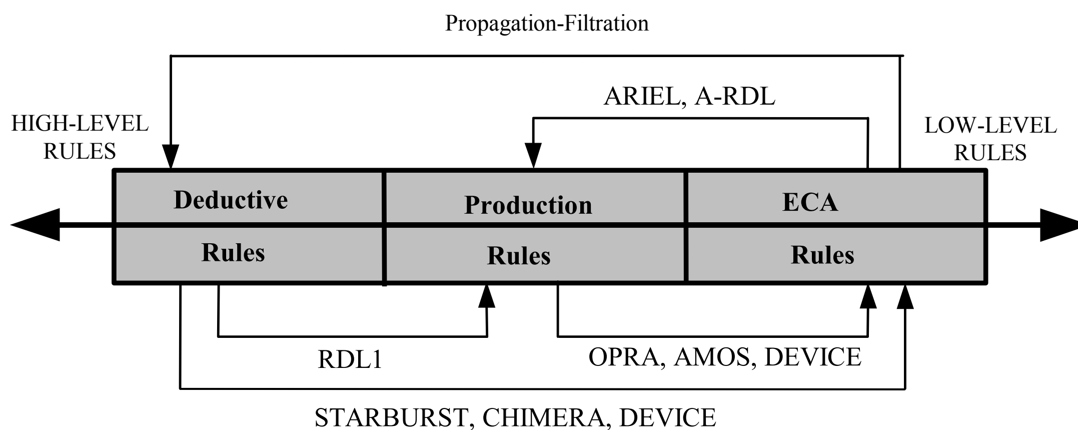

Figure 1. The rule spectrum

- facilitate applications beyond the scope of passive database systems;
- Perform tasks that require special-purpose subsystems in passive database systems.

The desired active behavior of ADBs is usually specified using *active rules*. There is a certain confusion about the term "active rules"; some researchers [4, 6] denote by this term the production rules met in expert system technology [7, 8], while others refer to the Event-Condition-Action (ECA) rules [9, 10, 11, 12] met in many active database systems.

Henceforth, we will use the term *active rules* to denote both these rule types collectively. Furthermore, we will use the above specific terms to address to each of the two active rule types in order to avoid the confusion:

- Production or "data-driven" rules are the rules of the form:

```
IF condition THEN action
```

The condition of these rules describes data states that should be reached by the database. When the condition is satisfied the production rule is fired (or triggered) and its set of actions is executed against the database.

- ECA or "event-driven" rules have the following form:

```
ON event IF condition THEN action
```

The ECA rule is explicitly triggered when the event of the rule has been detected, either in the database, caused by a data manipulation operator or externally by another system. The condition of the rule is only then checked and if satisfied the rule action is executed.

Typically, ADB systems support only one of the above two active rule types. However, there are few systems that support both ones.

1. ECA Rules

In the literature there are several ADB systems that support ECA rules. Most of them are object-oriented, such as HIPAC [10], SENTINEL [9], REACH [13], ADAM/EXACT [14, 11], SAMOS [12], AMOS [15], ACOOD [16], and NAOS [17]. ECA rules are the most "natural" choice for generic rule support since events conform to the message-passing paradigm of object-oriented computation and every recognizable message/method can be a potential event. Therefore, ECA rule execution can be very easily implemented as a "detour" from normal method execution.

Just before and/or right after method execution there is an opportunity to check if there is an event that should be monitored for this method and class. If there is, then the event occurrence is detected and signaled to the event manager of the system. Method execution proceeds normally between the two event detection phases. Therefore, event detection can be easily implemented as a side effect of the normal OODB method execution mechanism.

Events can be either database operations or happenings of interest external to the database, e.g. clock events or operating system events (interrupts). Furthermore, events can be both *simple* (*primitive*) and *complex* (*compound*). Complex events are combinations of simple events through event constructors, such as conjunction, disjunction, sequence, periodical, etc. Complex events are useful for integrating temporal aspects in an active database or for expressing complex logical conditions, as in SNOOP/SENTINEL [18], SAMOS [19], and ODE [20]. Furthermore, in section III we will show how complex events have been used in DEVICE to integrate production rules into an active database [21].

In relational databases, there are a number of different implementation techniques. This is mainly due to the fact that relational databases have a number of predefined generic operations that are common to all relations. Therefore, it would be quite inefficient to check for events every time a generic operation, such as insert or delete, is executed on any relation.

Among the first relational database systems to support ECA rules are POSTGRES [22] and STARBURST [23]. POSTGRES uses a tuple marking technique where each tuple that is a candidate to

trigger an ECA rule is permanently marked by a *rule lock* that indicates which rule will be triggered at run-time. Some times a rule lock is placed on the relation instead, when the granularity of the rule cannot be determined at rule-creation time or for space-saving purposes. At run-time, the tuple that is "modified" is checked for rule locks, and the appropriate rules are then executed.

STARBURST uses its extended features (such as *attachment procedures* which is a concept similar to *demons* in the technology of frame-based expert systems) in order to log the operations that trigger ECA rules. At the end of the transaction or at user-specified *checkpoints* the rule manager collects the triggered rules from the log and executes them.

Finally, A-RDL [24] and ARIEL [4] support events and ECA rules on top of production rules using delta-relations, a technique that will be described thoroughly in the next subsection.

a. Coupling modes. An important aspect of ECA rule execution is the exact time of event detection, condition checking, and action execution relative to the triggering operation and the end of the transaction. There are three possibilities for relative processing between event detection and condition checking (EC coupling) and between condition checking and action execution (CA coupling), called *rule-coupling modes*.

- *Immediate*. There is no delay between the evaluation/execution of the predecessor and successor ECA rule parts. For example, the action is executed immediately after the condition is satisfied.

- *Deferred*. The evaluation/execution of the successor ECA rule part is delayed until the end of the current transaction. For example, the condition of the rule is not checked after its event has been signaled but at the end of the transaction. This coupling mode may prove useful for e.g. checking integrity constraints, where many single updates violate the constraint but the overall effect is a valid transaction. If the condition is checked immediately after the first "illegal" update then a constraint violation will be detected, while if the check is delayed until the end of the transaction, the constraint violation might be repaired by following updates.

- *Decoupled.* The evaluation/execution of the successor ECA rule part is done in a separate transaction that might or might not depend on the current transaction. This mode is useful when long chains of rules are triggered and it is preferable to decompose it into smaller transactions to increase the database concurrency and availability.

A more detailed description of the concepts and features of ADBs can be found in the *Active Database Manifesto* [25]. Most of the above ADB systems can be found in an excellent collection of ADB research prototypes [26]. Here we have tried to introduce some of the concepts of active rules and present some implementation details about various active rule systems that will help our later discussion of multiple rule integration.

2. Production Rules

Several active relational database systems, such as RPL [27], RDL1 [28], DIPS [29], DATEX [30], and ARIEL [4] support production rules, in the fashion of OPS5-like expert systems.

All the above systems base their operation on the MATCH-SELECT-ACT cycle of production rule systems [7]. More specifically, production systems consist of a) the *working memory* (WM) that holds the initial data of a problem, plus the intermediate and final results and b) the *production memory* that holds the production rules. In analogy, the working memory of database production systems is the database itself while the production rules are kept in the system's rule dictionaries.

During the MATCH phase of the *production cycle* the system checks which rule conditions match with data in the working memory. A rule whose condition has been successfully matched against the working memory and its variables have been replaced by actual values is called *rule instantiation*. Production systems execute only one rule instantiation per cycle; therefore, when more than one rule instantiations are matched, they are all placed in the *conflict set* in order to be considered later for selection.

During the SELECT phase the system selects a single rule instantiation from the conflict set based on various *conflict resolution criteria*. Finally, the selected rule instantiation is executed in the ACT phase. The actions of the production rule may cause additional rule instantiations to be inserted or removed from the conflict set. The same procedure is continued until there are no more rule instantiations left in the conflict set after a MATCH phase.

One of the most important bottlenecks in the performance of production systems is the MATCH phase. The naive approach is to match all production rule conditions against all working memory elements at each cycle. However, various algorithms have been proposed that incrementally decide which rules should be added to or removed from the conflict set, such as RETE [8], TREAT [31], A-TREAT [4], GATOR [32], and LEAPS [30].

Almost all of the above algorithms are based on the compilation of the production rule conditions into a graph that is called *discrimination network.* The latter accepts in its input the modifications that occurred in the working memory and output the rule instantiations that should be added to or removed from the conflict set. The discrimination network usually maintains some information on the previously inserted elements in order to decide if the new elements combined with the previous ones make some rules match.

Most of the database production rule systems that we mentioned at the beginning of this section use some kind of discrimination network. More specifically, RPL uses a main-memory variation of RETE, while RDL1 uses a special petri net called *Production Compilation Network* [33]. DIPS system uses a novel, efficient rule condition matching algorithm that stores a “compressed” variation of the RETE network tokens into relational tables. Finally, ARIEL uses the A-TREAT algorithm, which uses virtual α-memories to save some space compared to TREAT along with special selection predicate indices for speeding-up the testing of selection conditions of rules.

In contrast, DATEX uses a complicated marking scheme [30], like POSTGRES, which employs a number of different indices to guide the search for matching first selection conditions and then to perform

joins to the appropriate direction of the condition. However, we believe that the same general principles apply to both the LEAPS algorithm and the discrimination network algorithms, and the only conceptual difference, in LEAPS, is that the discrimination network is not centralized but distributed across several persistent data structures. Of course, this distribution has certain benefits concerning the space and time complexity of the algorithm compared to the discrimination network algorithms. The price to be paid, however, is the increased compilation complexity and the inability to incrementally add new rules.

C. Active Knowledge Base Systems

In the previous subsection we have presented the integration of various rule types in various database systems. All rule paradigms are useful for different tasks in the database system. Therefore, the integration of multiple rule types in the same system is important. This will provide a single, flexible, multi-purpose knowledge base management system. Furthermore, such multi-rule systems are active because they support event detecting mechanisms.

In this subsection we present various techniques for unifying two or more different rule paradigms. More specifically, recall **Figure 1** from the previous subsection where the systems that attempt to integrate multiple rule types using a common framework are shown along with arcs that indicate which rules are the generic ones and which are emulated using the former. In this subsection we describe two major integration categories concerning ECA and production rules: a) integration of ECA rules in production rule systems, and b) integration of production rules into active database systems that support ECA rules only. In section IV the unification of production and deductive rule semantics is presented as an application of active knowledge base systems.

1. Integration of Events in Production Rule Conditions

ECA rules are low-level rules that describe explicitly their activation time. For example, the following rule does not allow any employees named 'Mike' which earn more than 500000 to be inserted to the relation emp:

```
ON   APPEND emp
IF   emp.name='Mike' and emp.sal>500000
THEN DELETE emp
```

Production rules, on the other hand, do not explicitly describe when they are activated. Instead, their declarative condition states that if somehow, at some point, the situation is met in the database, the rule is activated. Therefore, a generic difference between the event description of ECA rules and the condition of production rules is that the former describes a *change* in the state of the database while the latter describes a static database state.

In order to integrate events in the condition of production rules a new construct is needed to describe dynamic changes in the database instead of static conditions. This construct is *delta relations*. A delta relation consists of the tuples of a relation that have been changed during the current transaction or between rule checkpoints.

There are various delta relations for each normal database relation to reflect the various changes that can be applied to any given relation: a) for the tuples that have been inserted, b) for the deleted tuples, and c) for the tuples that have been updated. Delta relations are transient relations that hold data modifications during a transaction. After the transaction is committed these relations are flashed into their normal counterparts.

Using delta relations the ECA rule that has been presented at the beginning of this section can be expressed as the following production rule:

```
IF   e IN inserted_emp and e.name='Mike' and
     e.sal>500000
```

```
THEN DELETE e
```

The above rule can be used interchangeably with the ECA rule at the section beginning.

The technique of delta relations has been used by most systems that integrate events in production rules. For example, ARIEL [4] and A-RDL [34] are mainly production database rule systems that also support the use of ECA rules using delta relations. Of course, their approaches are slightly different from the one that has been described here.

ARIEL allows the definition of both production and ECA rules. However the conditions of either rule types cannot refer to the delta relations directly. Instead, delta relations are used by the low-level mechanism to "translate" the event into a condition reference to a delta relation. Of course, transition conditions can be expressed; i.e. the condition can explicitly refer to old and new values of a tuple.

A-RDL, on the other hand, does not allow the ECA rule syntax, i.e. events cannot be defined explicitly. It allows only the production rule syntax with explicit reference to delta relations, which is equivalent to event definition. Exactly the same concept is used in the integration of active and deductive rules using the Propagation-Filtration algorithm [35].

2. Integration of Production Rules in Active Databases

ECA rules are the most low-level rule type of the rule spectrum (**Figure 1**); therefore, they provide the most programming constructs for implementing add-on features with varying functionality in active databases. Production rules, on the other hand, are high-level programming tools with simple, declarative semantics, which is only a subset of the semantics that can be expressed with ECA rules. Of course, production rules in return are easier for a naive user to use than ECA rules.

The limited functionality of production rules can be easily "emulated" by ECA rules. The reason to do so is that a single system could provide both rule paradigms for different user categories.

There are two approaches to integrating production rules into ECA rules: the *multi-rule* and the *single-rule* approaches. Both are based on the compilation of a production rule into one or more ECA

rules. The ECA rules are then triggered by data modification events and they act accordingly in order to implement the semantics of production rules. In the rest of this subsection we present and compare these two production rule compilation techniques.

a. The Multi-Rule Approach. According to the multi-rule scheme, each production rule is translated into many ECA rules. Each ECA rule is triggered by a different, simple event, which is derived from a single condition element of the condition of the production rule. The condition of each ECA rule is almost the same as the condition of the production rule, minus the event.

This technique has been proposed both for production rules [36, 37] and deductive rules [38, 39]. Here we concentrate solely on production rules, while deductive rules are described in section IV.

Consider the following production rule:

P_1: `IF a & b & c THEN` <*action*>

where `a`, `b`, `c` are testing patterns for data items (tuples, objects, etc.) which we will be called, henceforth, just *data items* for brevity. Notice that these patterns can include variables, even shared among the patterns, which are not shown in this and the next rule examples. The above rule is compiled into the following 3 ECA rules:

EP_1: `ON insert(a) IF b & c THEN` <*action*>

EP_2: `ON insert(b) IF a & c THEN` <*action*>

EP_3: `ON insert(c) IF a & b THEN` <*action*>

The event `insert(x)` is a primitive event which is detected and signaled when the data item `x` is inserted in the database.

These three ECA rules suffice to monitor the database for the satisfaction of the condition of a production rule. The deletion of the data items `a`, `b`, `c`, need not be monitored since a conflict set that holds previously matched but not yet fired production rules does not exist. Therefore, the falsification of a previously satisfied declarative condition is indifferent.

b. The Single-Rule Approach. The single-rule integration scheme is based on the compilation of the condition of the declarative rule into a discrimination network that is built from complex events. The complex event network is associated with the event-part of an ECA rule. In this way the condition of the declarative rule is constantly monitored by the active database. The condition-part of the ECA rule is usually missing, except in some cases that will be mentioned later. Finally, the action-part of the ECA rule depends on the type of the declarative rule. This technique has been proposed for both production and rules [21, 40] that are described here and for deductive rules [41] that are described in section IV.

Following the single-rule compilation scheme, the production rule P_1 is translated into the following ECA rule:

```
SP1: ON    insert(a) & insert(b) & insert(c)
     [IF   true]
     THEN  <action>
```

where the operator & denotes the conjunction of the events.

The event manager of the ADB monitors individually the above primitive events. When each of them is detected its parameters are propagated and stored in the discrimination network, much alike the production systems. When more than one of them are detected their parameters are combined at the nodes of the network in order to detect the occurrence of the complex event *incrementally*. When, finally, the complex event is detected, the condition of the rule has been matched and the event manager forwards a tuple (or token) with the complex event's parameters to the rule manager which is responsible to schedule it for execution.

Notice that the incremental condition matching requires that when a primitive event occurrence is detected, then its parameters must be matched against the parameters of all previously detected event occurrences for the rest of the events, rather than only with the currently occurred ones. In order to achieve this, the parameters of all event occurrences are kept in the complex event network even after the end of the transaction. Actually, they are never deleted unless an explicit deletion is issued.

The single-rule approach corrects many of the problems associated with the multi-rule approach. In the following, these are discussed in detail.

Rule maintenance. In the multi-rule translation scheme, if someone wants to delete or temporarily disable a production rule, he/she must perform the same operation to all related ECA rules. However, this requires special care since the user might forget some of the ECA rules, and the rule base would then become inconsistent.

The single-rule approach avoids this problem by creating only one rule, which is maintained more easily. The de-activation of all the events (both simple and complex ones) associated with a deleted or disabled rule is automatically done by the system.

Redundant condition checking. Recall the production rule P_1 and the equivalent, according to the multi-rule translation scheme, 3 ECA rules EP_1-EP_3. Assume that the ECA rules have immediate EC coupling modes. We will examine what happens when the following sequence of events occurs, in the same transaction, in an empty database:

```
insert(c); insert(b); insert(a)
```

ECA rules are considered in the following order: EP_3, EP_2, EP_1. First EP_3 and then EP_2 are triggered but not executed since their conditions are not satisfied. Finally EP_1 is triggered, its condition is satisfied, and the action is executed. This behavior is correct since the production rule P_1 would have been fired under the same insertion sequence.

However, the above sequence of rule triggering creates performance problems since 3 ECA rules are triggered, and 6 condition elements are checked either successfully or not. Each of the 3 condition elements `a`, `b`, `c` is checked twice; the first time the check fails, while the second succeeds. This redundancy leads to poor performance, compared to the performance of the single-rule approach [40, 41] where each data item is checked only once.

Redundant action execution. Now re-consider the above event occurrence sequence but with the assumption that all 3 ECA rules have deferred EC coupling mode. This means that at the end of the

transaction all the ECA rules are triggered and executed because the data items have already been inserted by the time the rule conditions are considered. However, all 3 rules will execute the same action. This creates a problem because it is incorrect.

Of course, various conflict resolution strategies and/or priorities can be established at compile-time or during the design of the ECA rule base, in order to prevent the redundant execution of multiple rule actions. However, this solution complicates things further because these conflict resolution strategies must be enforced separately from the conflict resolution criteria that are based on semantics.

The single-rule approach avoids this problem by having a single rule. Furthermore, the DEVICE system that will be presented in the next section has a centralized rule manager that resolves conflicts among multiple production rules, allowing only one to fire according to various conflict resolution criteria that are based on the semantics of the application.

Net effect. One more problem associated with the immediate EC coupling mode is the absence of the *net effect* of events. When an event triggers a rule and that rule is selected for execution, there is no way to "undo" the rule activation by reversing the effect of the triggering event. For example, when the creation of the object activates a rule, the rule is going to fire even if the object is deleted before the end of the transaction.

This problem exists for the rules with immediate EC coupling, even if the underlying active system does support net effects, because rules are immediately activated without waiting for the end of the transaction. The immediate mode is simply not compatible with the state description nature of production rule conditions.

In the case of immediate EC and deferred CA coupling modes, in order to overcome the absence of net effects the condition is re-checked just before the action of the ECA rule is executed. In this way it is assured that the event and the condition that triggered the rule after the event signaling is still true at the end of the transaction. For example rule EP_1 would look under this scheme as follows:

```
ON    insert(a)
```

```
IF    b & c
THEN  (IF a & b & c THEN <action> ELSE true)
```

In the case of deferred EC and CA coupling, the check should be included only in the condition:

```
ON insert(a) IF a & b & c THEN <action>
```

However, the above solution would incur overhead on the performance of rule execution because it duplicates checking of already checked conditions. The single-rule approach avoids this problem of net event effects by delaying the execution of triggered rules until the end of the transaction.

III. DEVICE: An Active Object-Oriented Knowledge Base System

In the previous section we have presented various techniques for unifying two or more different rule paradigms. Among the techniques presented was the single-rule translation scheme, which integrates production and deductive rules into an active database system that generically supports only ECA rules. In this section, we present in detail an active object-oriented knowledge base system, called DEVICE [21, 40, 41], which uses the single-rule translation approach.

In the following, we first describe the architecture and the production rule language of the DEVICE system. Then the operational semantics of production rules in DEVICE are described along with their integration with ECA rules. The details of compiling the production rule conditions into complex event networks are presented separately in order to make clear how rule conditions are incrementally matched at run-time. In the next section we present deductive databases as an application of DEVICE by implementing deductive rules on top of production rules.

A. System Architecture

The overall architecture of the DEVICE system is shown in **Figure 2**. DEVICE mainly consists of classes and meta-classes, which are introduced to the core active OODB system and extend its functionality. More specifically, DEVICE consists of two major components: compile-time and run-time modules.

The compile-time modules of DEVICE are mainly meta-classes that host the methods for compiling the production rule into a network of complex events plus one ECA rule using the single-rule translation technique we described in the previous section.

The run-time modules of DEVICE are various OODB classes that host the structure and behavior of complex events and production rules. They are usually referred to as *managers*, such as the complex event manager, the production rule manager, etc. Rules and events are first-class objects, instances of the corresponding managers.

The ECA rule manager is the most generic one and is part of the core active OODB system. The production rule manager is a subclass of the ECA rule manager. It partly inherits the functions of the former as well as it re-defines many of them in order to capture the higher-level semantics of production

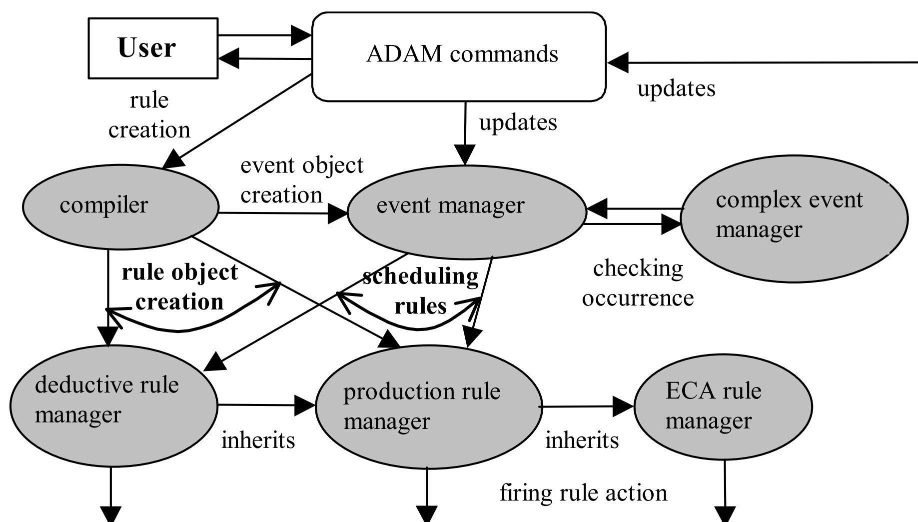

Figure 2. The architecture of the DEVICE system

rules.

Complex events are subtypes of generic OODB events. Complex events are the building components of the discrimination network which is used to match the production rules' conditions. The event manager keeps track of which simple events have occurred and combines them incrementally to match the rules' conditions.

The DEVICE system is fully implemented on top of ECLiPSe Prolog as an extension to the active OODB EXACT [14], which is an extension of ADAM OODB [42]. DEVICE is an extensible system as it is proved in the next section where the implementation of deductive rules on top of production rules is described.

B. The Production Rule Language

This section describes the system's declarative rule language which follows, for the most part, the OPS5 [7] paradigm influenced by the OODB context of DEVICE. Production rules are expressed as a *condition*, which defines a pattern of objects to be detected over the database, followed by an *action* to be taken.

The condition of a rule is an *inter-object* pattern, which consists of the conjunction of one or more (positive or negative) *intra-object* patterns. The intra-object patterns consist of one or more *attribute* patterns. For example, the following rule condition defines an employee working in the 'Security' department but his/her manager is different from the department's manager:

```
PR1: IF    E@emp(dept:D,manager:M) and
           D@dept(name='Security',manager\=M)
     THEN  delete ⇒ E
```

The first of the above intra-object patterns denotes an instance `E` of class `emp`. The second intra-object pattern describes the department `D` of employee `E` whose `name` attribute is equal to 'Security' and its `manager` attribute is different from the manager `M` of `E`.

Variables in front of the class names denote instances of the class. Inside the brackets, attribute patterns are denoted by relational comparisons, either directly with constants or indirectly through variables. Variables are also used to deliver values for comparison to other intra-object patterns (joins) in the same condition or to the action part of the rule. The values can be both object references and normal values, e.g. integers, strings.

We notice here that the condition of PR_1 can be written also as:

```
E@emp(name.dept='Security',manager:M,manager.dept\=M)
```

Attribute patterns can navigate through object references of complex attributes, such as the complex attribute `name.dept`. The innermost attribute should be an attribute of class `emp`. Moving from right to the left of the expression, attributes belong to classes related through object-reference attributes of the class of their predecessor attributes. We have adopted a right-to-left order of attributes, contrary to the C-like dot notation that is commonly assumed following the functional data model of the core OODB system ADAM [43]. Under this interpretation, the chained "dotted" attributes can be seen as function compositions.

During a pre-compilation phase, each rule that contains complex attribute expressions is transformed into one that contains only simple attribute expressions by introducing new intra-object patterns. The above pattern is actually transformed into the condition of PR_1.

There can also be negated intra-object patterns in the condition. A negated intra-object pattern denotes a negative condition that is satisfied when no objects in the database satisfy the corresponding positive intra-object pattern. Notice that only safe rules are allowed. The following rule condition identifies an employee who has worked more hours than anyone.

```
PR2: IF    E1@emp(hours_worked:H,salary:S) and
           not E2@emp(hours_worked>H) and
           prolog{S1 is 1.1*S}
     THEN  update_salary([S,S1]) ⇒ E1
```

The use of arbitrary Prolog or ADAM goals to express some small static conditions or to compute certain values is allowed in the condition through the special `prolog{}` construct. In the appendix, we include the full syntax of the condition-part language.

The action part of a production rule defines a set of updates to be performed on the database objects that were identified in the rule condition. These updates are expressed in an extended Prolog language, which includes the default, procedural data manipulation language of ADAM. The syntax of the ADAM messages can be found in [43].

Examples of production rule actions are given in rules PR_1 and PR_2 above. In PR_1, a 'Security' employee is deleted when his/her manager is different from the department's manager whereas, in PR_2, the harder worker's salary is increased by 10%.

C. Integration of Production Rules

Production rules are integrated in the active database system following these steps:

1. The condition of the rule is compiled into a discrimination network that consists of complex events;
2. The last event in the network is the triggering event of the ECA rule;
3. The condition part of the ECA rule is usually *true* because all condition tests have been incorporated into the complex event. However, if the `prolog{}` construct is present, then the Prolog goals are incorporated in the condition of the ECA rule;
4. The action part of the ECA rule is the same as the production rule action.

At run-time, the active database system monitors the simple events that have been created for the production rules. When a simple event is detected it is signaled to the event manager who is responsible for propagating its parameters to the complex event network. The event parameters are propagated through tokens, which are tuples that constitute of pairs of condition variables and their values.

Tokens can be positive or negative depending on the initial simple insertion or deletion event that has been detected. If a token is propagated through the whole complex event network, it means that the corresponding rule has been either matched (in the case of positive tokens) or unmatched (in the case of negative tokens). The rule along with the last event's parameters is called rule instantiation, and is forwarded to the production rule manager in order to schedule it for execution.

The production rule manager receives all the detected complex event occurrences from the event manager and selects those events that activate production rules. The positive rule instantiation tokens are placed into the “conflict set”. The negative tokens cause the corresponding positive rule instantiations to be removed from the conflict set, if they still exist there.

When multiple rule instantiations are placed in the conflict set, there is an ambiguity concerning the number and order of rules to be executed. The OPS5 approach applies heuristic strategies to select a unique rule instantiation to execute [7]. The active database systems’ approach uses priorities to resolve the rule execution order. In DEVICE, the OPS5 conflict resolution heuristics have been incorporated into the priority mechanism of the active OODB system. The application of any of the heuristics is controlled by an appropriate class variable of the rule manager that can be set to `on` or `off`.

The conflict set is a Prolog list (LIFO structure) that is stored as a class attribute in the production rule manager. The *refractoriness* criterion removes the rule instantiation tokens that have been executed from the conflict list. The *recency* criterion inserts the newly derived rule instantiations at the beginning of the conflict list, in order to be considered before the older ones.

Finally, the *specificity* criteria selectively picks-up at run-time from the conflict set rule instantiation tokens that their conditions are more specific than the others. The specificity of a rule is determined by the number of event objects involved during condition matching and is calculated at compile-time by counting the total number of generated events for the condition.

After the rule manager selects a rule instantiation for execution, the condition part of the rule is checked. Usually the trivial `true` condition is associated with DEVICE rules unless the `prolog{}`

construct is present at the rule definition. If the condition evaluates to false, then the rule is not fired. If the condition is confirmed, then the action part of the rule must be scheduled for execution. The action is executed as a compound Prolog goal using the immediate CA coupling mode.

In DEVICE, rule selection and execution are initiated either at the end of the transaction or at intermediate user-specified *checkpoints*. After the first rule instantiation is selected and executed, the rule manager self-raises a checkpoint in order to continue with the next production cycle by considering all the previous rule instantiations plus any new ones that have been produced by the execution of rule actions. This cycle continues until a *fixpoint* is reached where there are no more rule instantiations left in the conflict set. This happens when rule actions either do not produce new rule instantiations or evoke explicit object deletions that propagate up to the conflict set. After the fixpoint is reached, the control of the transaction is given back to the user.

The *net effect* of events is guaranteed by the deferred EC coupling mode. When two events of the same transaction cause contradictory (a positive and a negative) rule instantiation placements in the conflict set, then the rule instantiation is eliminated from the conflict set before the rule selection and execution sequences begin at the end of the transaction. Therefore, no rule is executed. When the two events above are issued at different transactions but the rule instantiation in question has not yet been selected for execution, a similar net effect is produced.

1. Compilation and Matching of Rule Conditions

The efficient matching of production rule conditions is usually achieved through a discrimination network. DEVICE smoothly integrates a RETE-like discrimination network into an active OODB system as a set of first class objects by mapping each node of the network onto a complex event object of the ADB system. This section overviews both the structure and behavior of the network. More details about both the compilation and run-time aspects of the network nodes can be found in [21, 40].

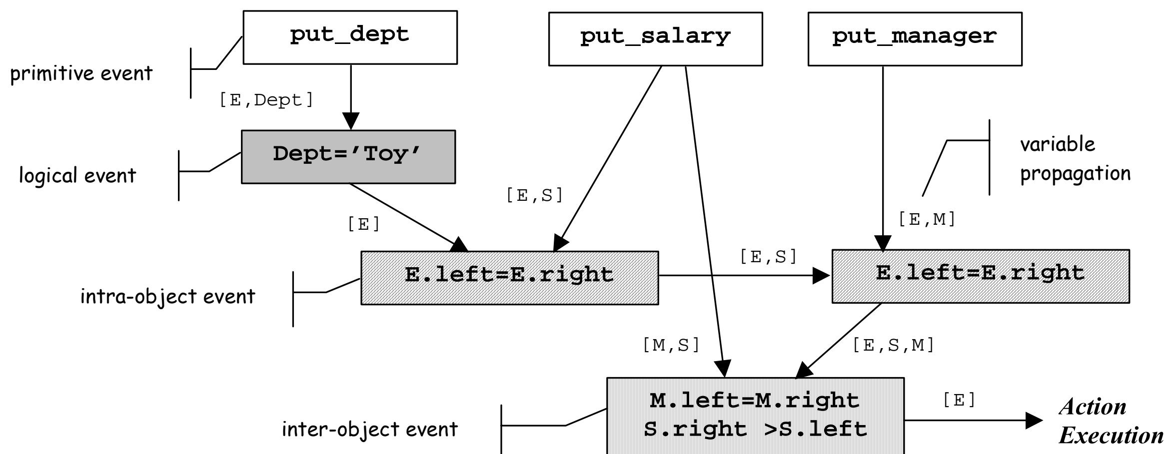

Figure 3. A sample complex event network

The complex event network consists of 3 event types: primitive, logical, and two-input events. Throughout this section, we describe the structure and behavior of these event types using the following example whose complex event network is shown in Figure 3.

```
PR3: IF    E@emp(dept='Toy',salary:S,manager:M) and
           M@emp(salary<S)
     THEN delete ⇒ E
```

a. Primitive events. The DEVICE network has multiple input sources, which are the primitive database events detected by the active database system. Each attribute pattern inside any intra-object pattern in the condition is mapped on a primitive event that monitors the insertion (or deletion) of values at the corresponding attribute. In **Figure 3**, there are several primitive events, such as `put_salary`, `put_manager`, etc., and their counterpart of type `delete_`, which are not shown for simplicity.

The *signaling* of a `put_` type primitive event denotes that a certain database state has been reached by inserting data in the database. On the other hand, the occurrence of `delete_` type events denotes that a certain pattern in the rule condition that was previously present in the database is no longer valid. To model such semantics, *anti-signaling* is used. We notice that `update_` type events are emulated by anti-signaling a `delete_` type event followed by the signaling of a `put_` type event.

When primitive events are signaled (or anti-signaled), the event manager forwards a positive (or negative) token with the message parameters to the successor network. Each network object internally processes the input tokens and checks if a complex event can be signaled according to the current input signal and the local history of event activation. When appropriate, output tokens are forwarded further in the event network.

b. Logical events. Logical events perform simple attribute tests, and they are only raised when the associated condition is satisfied. In DEVICE, logical events map attribute comparisons with constants, and they are signaled by primitive events to perform a check on their parameter. If the check is successful, an output token is propagated to a successor event in the event network. Logical events are the equivalent of α-memories of the RETE network. In **Figure 3**, there is one such logical event for the attribute test against the constant 'Toy'.

c. Two-input events. An intra-object pattern that consists of at least two attribute patterns is translated into a two-input event (also called *intra-object event*) that joins the parameters of the input events (primitive and/or logical) based on the object identifier (OID) of the message recipient objects. In **Figure 3**, there are two intra-object events. The intra-object event that joins the first two attribute patterns is further joined with the third attribute pattern into a new intra-object event. Should more attribute patterns exist this procedure goes on until all the attribute patterns are catered for.

Multiple intra-object patterns are mapped into multiple intra-object events that are joined in pairs based on the shared variables between the intra-object patterns in the rule condition. These events are called *inter-object events*. In **Figure 3**, there should be two inter-object events. The first should join the two intra-object patterns on the value of variable M. The second should join the previous inter-object event with the second intra-object pattern on variable S. However, these two joins are simultaneously performed in the same inter-object event for optimization. Furthermore, we notice that the second inter-object pattern consists of only one attribute pattern, thus instead of an intra-object event, the intra-object pattern is represented in the network by a primitive event.

The last inter-object event of the network represents the whole rule condition, and it is directly attached to the ECA rule that maps the original rule.

Intra-object and inter-object events are collectively called two-input events and are treated in a uniform way. Here they have been analyzed separately for mere presentation purposes. Two-input events are the equivalent of β-memories of the RETE network.

Two-input events receive tokens from both inputs whose behavior is symmetrical. The positive incoming tokens are permanently stored at the corresponding input memories and are joined with the tokens of the opposite memory. The join produces positive output tokens (one or more) according to a pre-compiled pattern and are propagated further to the event network.

d. Token deletion. Tokens describe database states, and they persist inside the two-input event memories beyond the end of the transaction. They can be only explicitly deleted to reflect deletions in the database. The deletion of tokens is triggered by the propagation of anti-signals in the network.

When a two-input event receives a negative token at one of its inputs, it deletes it from the corresponding memory and a negative token is output. The output token contains elements only from the deleted (incomplete) token because there is no need to join it with the tokens of the right memory, unless the two-input event is the last in the network [21].

e. Negation. Negative intra-object patterns denote conditions that should not be met in order for the whole rule condition to become true. The negative patterns are treated much the same as the positive ones except that the inter-object event whose input corresponds to the negative pattern is a *negative event*.

Structurally, negative events do not differ from positive ones. However, their behavior is different because the detection of the intra-object event at the negative input indicates that the (negative) inter-object event does not occur and vice-versa. Another difference of negative events is they behave differently depending on the input source: the “negative” or the “positive” inputs. The “negative” input does not contribute values to the output token of the inter-object event because the negation is not

constructive and only stands for value testing (safety requirement). More details on negative two-input events can be found in [40].

IV. Applications of Active Knowledge Base Systems

In this section we present two applications that were implemented on the DEVICE system, namely deductive databases and data warehousing. The former is based on the emulation of deductive rules on top of production rules. The latter is based on the integration of DEVICE with InterBase*, a multi-database system.

A. Deductive Databases

Deductive databases [2, 1] incorporate aspects of logic programming into databases and thereby they bridge the gap between databases and knowledge bases. Deductive databases allow users, through the means of deductive rules, to deduce facts concerning data stored in the database.

A deductive rule consists of the *head* (also called conclusion or consequent) and the *body* (also called condition or antecedent). The interpretation of a deductive rule is that *if the condition of the rule is satisfied, then the objects described by the head of the rule should be in the database*.

The body of a DEVICE deductive rule is identical to the condition of a production rule. The head or conclusion is a *derived class template* that defines the objects that are derivable when the condition is true. An example of a couple of DEVICE deductive rules is the following:

```
DR1: IF    P@path(edgeBegin:X,edgeEnd:Y)
     THEN  arc(edgeBegin:X,edgeEnd:Y)
DR2: IF    A@arc(edgeBegin:X,edgeEnd:Z) and
           P@path(edgeBegin:Z,edgeEnd:Y)
     THEN  arc(edgeBegin:X,edgeEnd:Y)
```

The deductive rules above define the transitive closure of the connectivity between any two vertices in a graph. Class `arc` is a derived class, i.e. a class whose instances are derived from deductive rules. Only one derived class template is allowed at the *head* of a deductive rule. However, there can exist many rules with the same derived class at the head. The final set of the derived objects is a union of objects derived by all the rules that define the derived class. The derived class template consists of attribute-value pairs where the value can either be a variable that appears in the condition or a constant. The syntax is given in the appendix.

1. Common Semantics for Production and Deductive Rules

The integration of deductive rules in the DEVICE system is achieved by mapping the deductive rule semantics on production rules. The RDL1 system [28, 33] made an important contribution to the unification of production and deductive rule semantics. More specifically, the production rule language of RDL1 has been proved to be as expressive as Datalog with negation [33].

The condition of an RDL1 production rule is a range restricted formula of the relational calculus, as in Datalog, while the action can be a set of positive or negative literals. A positive literal means the insertion of the corresponding tuple in the database while the negative literal means deletion. In contrast, Datalog allows only a single positive literal in the head, which is equivalent to the RDL1 rule action.

According to the semantics of deductive rules, as described by Widom in [5], when the condition of the deductive rule is satisfied, then the tuple/object described by the rule head “is in the relation” of the head's predicate. There can be two interpretations, according to the materialized and the non-materialized approaches to deductive databases.

If the derived relation/class is materialized then the derived tuple/object must be inserted in the database (procedural action). Otherwise, according to the non-materialized approach, the derived tuple/object is inserted in the answer set of the query that evoked the rule-processing algorithm. We can safely consider that the answer set is a temporarily materialized derived relation, which is deleted after the

answer to the query. Therefore, for both approaches, the operational semantics of the bottom-up processing of deductive rules can be compared to forward chaining production rules.

Thus, production and deductive rules differ only in their consequent/action part while the condition part is a declarative query over the database for both. The action part of a production rule is an explicit set of procedural database modifications while the consequent part of a deductive rule is an implicit action of object creation.

Deductive rule compilation is a little more complex than the above simple scheme. For example, consider the following deductive rule:

```
D1: IF a & b THEN d
```

which (according to the equivalence of deductive and production rules) is translated into the following ECA rule, using the single-rule approach of DEVICE:

```
SD1: ON    insert(a) & insert(b)
     [IF   true]
     THEN  insert(d)
```

or into the following 2 ECA rules, using the multi-rule approach of Ceri and Widom [38]:

```
ED1: ON insert(a) IF b THEN insert(d)
ED2: ON insert(b) IF a THEN insert(d)
```

The above rules only monitor the insertion of condition data items. However deductive rules must also monitor the deletion of condition items in order to keep the database consistent. If for example item `b` is deleted from the database then item `d` can no longer exist in the database, therefore it must be deleted.

The multi-rule approach of Ceri and Widom [38] extends the rule set ED_1-ED_2 with the following two ECA rules:

```
ED3: ON delete(a) IF b THEN delete(d)
ED4: ON delete(b) IF a THEN delete(d)
```

where the event `delete(x)` monitors the deletion of the `x` data item.

Furthermore, this approach (called *delete-and-re-derive*) requires one more rule to check and re-insert some deleted derived objects due to possible alternative derivations:

```
ED5: ON delete(d) IF a & b THEN insert(d)
```

The multi-rule approach of Griefahn and Manthey [39], on the other hand, avoids the unnecessary deletions in the first place by incorporating a check in the condition of the "deletion" rules about the alternative derivations:

```
ED'3: ON delete(a) IF b & ¬d THEN delete(d)
```

Notice that the `¬d` will be re-evaluated based on the deductive rule definition in the new state that the database has come after the deletion of `d`.

The single-rule approach of DEVICE extends ECA rule SD_1 with an ELSE part, which is executed when the condition of the original deductive rule is falsified due to the deletion of one or more of the data items:

```
SD'1: ON   insert(a) & insert(b)
      [IF  true]
      THEN insert(d)
      ELSE delete(d)
```

Furthermore, a counting mechanism, that was introduced by Gupta et al. in [44], is used in order to check if the derived object that is about to be deleted has alternative derivations.

2. Implementation of Deductive Rules in DEVICE

As it was noticed in the previous section the simple translation scheme of production rules is not adequate to fully capture the semantics of deductive rules. There are certain extensions that should be made: a) the anti-action or ELSE part, and b) the counting mechanism.

In order to model the deletion of a derived object, production rules are extended with an `anti_action` (or ELSE) part that hosts the derived object deletion algorithm. Using this extended scheme, a deductive rule can be modeled by a single production rule if the positive action is mapped to the `action` part of the rule, and the negative action is mapped to the `anti_action` part:

```
IF   condition
THEN create(object))
ELSE delete(object))
```

Furthermore, the rule manager should be extended in order to be able to execute the anti-action rule part upon the receipt of a negative token from the event manager. Therefore, the semantics of deductive rules are implemented under a new deductive rule manager that is a subclass of the production rule manager. The former inherits a part of the common behavior from the latter and overrides some of the structural and behavioral features of the latter.

Concerning the multiple derivations problem, before a derived object is removed from the database it must be ensured that it is not deducible by another rule instantiation. For this reason, a counter mechanism, which stores the number of derivations of an object [44], is used. If the derived object has a counter equal to 1, then it is deleted; otherwise 1 is subtracted from the counter.

Furthermore, the creation of a new derived object should only be done if the object does not already exist, otherwise two distinct objects with the same attribute values will exist. This is a consequence of the generic differences between an OID-based OODB and a value-based deductive database [45]. When a derived object already exists, then its counter is just increased by 1.

The above operational semantics is modeled by the following extended production rule which is translated into an ECA rule using the procedure described at the beginning of this section:

```
IF   condition
THEN (IF exists(object)
       THEN inc_counter(object)
       ELSE create(object))
```

```
ELSE (IF counter(object)>1
       THEN dec_counter(object)
       ELSE delete(object))
```

The conflict resolution strategies of deductive rules differ from production rules. The recency strategy is not used and, therefore, new rule instantiations are appended to the conflict set. The rule search space is, thus, navigated in a breadth-first manner in order to model the set-oriented semi-naive evaluation of deductive rules [1].

Specificity is overridden by the *stratification* control strategy, which ensures that the derivation process has not infinite loops due to recursion and negation. When a deductive rule is created, the derived classes that appear in the condition are collected along with their strata (i.e. order of derivation). The algorithm presented in Ullman [1] checks if the new rule, along with the existing ones, constitute a stratified logic program and modifies their strata as a side effect. The strata define a partial ordering of rules, which is used to resolve rule selection at run-time using exactly the same algorithm as for specificity.

B. Data Warehousing

A *Data Warehouse* is a repository that integrates information from data sources, which may or may not be heterogeneous and makes them available for decision support querying and analysis [46]. There are two main advantages to data warehouses. First, they off-load decision support applications from the original, possibly on-line transaction, database systems. Second, they bring together information from multiple sources, thus providing a consistent database source for decision support queries.

Data warehouses store *materialized views* in order to provide fast and uniform access to information that is integrated from several distributed data sources. The warehouse provides a different way of looking at the data than the databases being integrated. Materialized views collect data from databases into the warehouse, but without copying each database into the warehouse. Queries on the warehouse can then be answered using the views instead of accessing the remote databases. When

modification of data occurs on remote databases, they are transmitted to the warehouse. Incremental view maintenance techniques are used to maintain the views consistent with the modifications.

The deductive rules of DEVICE that were described in the previous subsection provide an excellent framework under which materialized views can be implemented. However, DEVICE supports data only from its own internal database therefore a new tool that integrates and maintains data from heterogeneous databases should be built. This system is called InterBase^{KB} [47] and is based on the integration of DEVICE and a multi-database system, called InterBase* [48].

1. System Architecture

The architecture of the InterBase^{KB} system is shown in Figure 4. The InterBase^{KB} system extends the InterBase* multi-database with a KB module (KBM) that is responsible for integrating the schema of the component database systems and for running the inference engine that materializes the views of the component databases inside the data warehouse. The components of the InterBase^{KB} system are the following:

a. InterBase^{KB} Server. This server maintains data dictionaries and is responsible for processing InterSQL

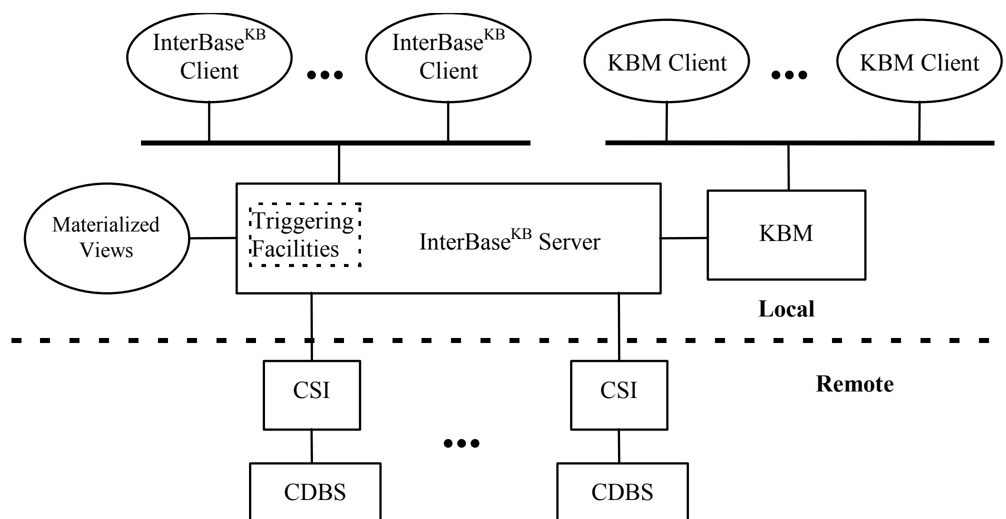

Figure 4. The Architecture of the InterBase^{KB} System

queries, as in the InterBase* system. Furthermore, it hosts the materialized views of the data warehouse. This means that the users of the data warehouse need not access the base data of the CDBSs but can instead directly access the views provided for them inside the warehouse. The server does not host the global integrated schema because this is defined and maintained inside the KB module, whose capabilities of inference and data modeling are a superset of the capabilities of InterBase^{KB} server. However, if the administrator of the data warehouse chooses to materialize the integrated view of the CDBS base data, then these will be stored at the InterBase^{KB} server's database, and the integrated global schema will be hosted by the server.

The InterBase^{KB} server extends the InterBase* server with triggering capabilities. This means that when an InterBase^{KB} or a KBM client inserts, deletes or updates data in the InterBase^{KB} server's database, an event is raised that signals the occurrence of such a data modification action. This event is communicated to the KBM and possibly triggers an active or some declarative rule.

On the other hand, modifications to the data of the CDBSs are not captured by the triggering system of the InterBase^{KB} server but are handled by the CSIs. However, the changes that are detected at the CSIs level are propagated to the triggering subsystem of the InterBase^{KB} server, which is responsible for delegating it to the KBM for further processing

b. InterBase^{KB} Clients. These are the clients of the old non-federated multi-database system InterBase* and are kept to support the old applications. They connect to the InterBase^{KB} server and issue InterSQL [49] queries against the component databases or the materialized views of the data warehouse which are stored inside the InterBase^{KB} server's database. They cannot be connected to the KBM because InterSQL cannot be translated to the fully object-oriented programming language of the KBM.

c. Knowledge Base Module (KBM). This module includes an active OODB, extended with declarative rules and an inference engine for a) integrating the schemes of the component databases and b) defining and maintaining the materialized views of the data warehouse. The architecture of the KBM is shown in Figure 5. The components of the KBM are the following:

The Active Knowledge Base (A-KB) core. The KBM's core is the active object-oriented knowledge base system DEVICE (see previous section). The A-KB is responsible for a) integrating the schemes of the component databases, b) defining and maintaining the materialized views of the data warehouse (stored at the InterBase^{KB} server), and c) propagating updates of the materialized views to the data sources.

The A-KB core communicates with the rest of the InterBase^{KB} system through a number of interface components. The administrator of the warehouse directly communicates with the A-KB core and can evoke methods for creating/destroying, enabling/disabling declarative rules for integrating the schemes of the component database systems and defining materialized views.

The OO-InterSQL interface. This interface translates the first-order rule definition language of A-KB into relational commands of InterSQL. Furthermore, it is responsible for translating simple object accessing methods into SQL retrieval/modification operations.

The Triggering Interface. This interface is responsible for capturing any data modification events trapped by either the triggering subsystem of the InterBase^{KB} server or the component system interfaces. The latter are not communicated directly to the KBM, but through the triggering subsystem of the InterBase^{KB} server. Therefore, the triggering interface of the KBM needs to capture only one event format. The events

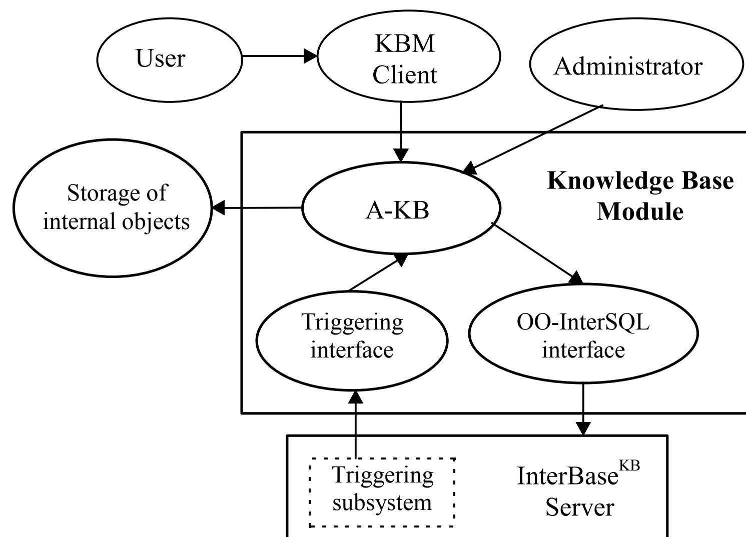

Figure 5. The Architecture of the Knowledge Base Module

raised by the component system interfaces denote changes at the base data of the data sources while the events raised by InterBase^{KB} server denote changes made by the InterBase^{KB} or the KBM clients to the materialized views stored at the warehouse.

d. KBM Clients. These clients have to be used in order to access the extended features of InterBase^{KB}, like global integrated schema, updateable materialized views, purely object-oriented database programming language, and declarative rules for programming expert database applications. This simple client accepts user queries interactively or user programs in batch mode and forwards them through the network to the KBM. The language used is Prolog extended with object-oriented and persistence features, like OIDs, messages, etc.

e. The Storage System. The KBM needs to store data and methods, both for the user and for internal purposes, such as rule and event objects, the discrimination network memories, etc. The storage system is based on the built-in storage facilities of the underlying Prolog system, which is either ECLiPSe or SICStus Prolog.

f. Component Database Systems (CDBSs). These are the heterogeneous systems that are integrated into the multi-database. Furthermore, they are the data sources for the data warehouse.

g. Component System Interfaces (CSIs). These components act as an interface for the InterBase^{KB} server to the heterogeneous CDBSs. They translate InterSQL queries and commands to the native query language of the CDBS, and translate back the results, therefore they are version specific. While this is adequate for InterBase*, in InterBase^{KB} it is necessary for the interfaces to be able to detect changes of base data that have occurred inside the CDBSs by their native users and inform the InterBase^{KB} and the KBM subsequently that the data warehouse views might be inconsistent. It is the task of the KBM to decide and propagate these changes to the InterBase^{KB} server's database. However, it is the responsibility of the interface to detect the changes.

There are several ways to detect the data changes at the data sources, depending on the nature of the source itself. If the data source is a full-fledged database system, then the following solutions can be used:

- If the database system supports *triggering* or *active rule* facilities, these can be directly programmed through the CSI to directly notify data changes of interest.
- If the data source lacks active rule facilities, the next alternative is to inspect periodically the *log files* of the CDBSs to extract any interesting events.
- If the database system lacks both of the above features, the CSI can be programmed to periodically query the CSDB (polling) to detect any changes that have occurred since the last query. This can be very inefficient if the polling period is too low or very inaccurate if the polling is done infrequently and important changes are discovered too late.
- Finally, if the data (or information) source is not a database system but an application or a utility, periodic snapshots of the data can be provided and incrementally compared to detect the changes.

Regardless of the way the changes of data at the sources are detected, the communication of those changes to the data warehouse can either be done as soon as the change is detected or periodically. The latter solution can be configured to send the changes at the data warehouse when the latter is off-line, i.e. when it is not used for large decision support queries but runs in a maintenance mode. In this way, the maintenance of materialized data does not clutter the data warehouse during its normal operation.

2. View Materialization and Maintenance

Schema integration in multi-database and heterogeneous environments is usually achieved by defining common views of the underlying data. In this way, details of the heterogeneous data sources are abstracted away, and the user transparently sees a global schema. The view definition language and view materialization mechanism of InterBase^{KB} is provided by the deductive rules of the A-KB core (namely

DEVICE), which has been described in section IV.A. More specifically, each derived class plays the role of a view class whose definition is included into the set of deductive rules that have the derived class as their head. Details concerning the type of the view class and the update of the views are reported elsewhere [47].

An important advantage of using a discrimination network for the incremental maintenance of materialized views is that the views are self-maintainable [50]. This means that, in order to derive what changes need to be made to the materialized views in the warehouse when base data are modified, there is no need to query back the data sources [51]. All the necessary past data needed for maintaining the view is kept inside the memories of the two-input complex events.

3. Integration of Heterogeneous Data

In this section, we describe the mechanisms of InterBase^{KB} for integrating data from heterogeneous data sources.

a. Schema translation of the component databases. The various component databases or data sources probably have their own schemata, which might have been expressed in different data models. Therefore, a mechanism is needed to translate the data model of each data source to the common data model of the data warehouse. InterBase^{KB} supports an object-oriented common data model [52], which is rich enough to capture the heterogeneity between the data models of the data sources.

b. Resolution of schematic and semantic conflicts. After the homogenization of the data models, there is still a need to resolve the conflicts among the schemata of the data sources. There can be many kinds of conflicts among the local schemata [52, 53, 54], such as schematic, semantic, identity, and data conflicts. The mechanism for schema integration should be general enough to be able to resolve most of them.

Database	***company_A***	***company_B***
class	inventory	inventory
attributes	dept: deptID item: string quantity: integer	dept: deptID $item_1$: integer $item_2$: integer ... $item_n$: integer

Database	***company_C***			
class	$item_1$	$item_2$	**...**	$Item_n$
attributes	dept: deptID quantity: integer	dept: deptID quantity: integer		dept: deptID quantity: integer

Table Σφάλμα! Άγνωστη παράμετρος αλλαγής.**.** Schemata of company databases

In the following example we demonstrate a schema integration problem (Table Σφάλμα! Άγνωστη παράμετρος αλλαγής.) and we also provide the $InterBase^{KB}$ solution (Figure 6).

Consider a federation of company databases in a corporation, consisting of OODBs company_A, company_B and company_C, corresponding to each of the three companies A, B and C. Each database maintains information about the company's inventory. The schemata of the three databases are shown in Table Σφάλμα! Άγνωστη παράμετρος αλλαγής..

The company_A database has a single class inventory which has one instance for each department and each item. The database company_B also has a single class inventory but items appear as attribute names whose values are the corresponding quantities. Finally, company_C has as many classes items and each instance represents each department and the corresponding item quantity.

```
DBA: IF     I@inventory/company_A(dept:D,item:T,quantity:Q)
     THEN   inventory(dept:D,item:T,quantity:Q)

DBB: IF     I@inventory/company_B(dept:D,T\=dept:Q)
     THEN   inventory(dept:D,item:T,quantity:Q)

DBC: IF     I@T/company_C(dept:D,quantity:Q)
     THEN   inventory(dept:D,item:T,quantity:Q)
```

Figure 6. Deductive rules for integrating schemata

```
DB'A:  IF     I@inventory_company_A(dept:D,item:T,quantity:Q)
       THEN   inventory(dept:D,item:T,quantity:Q)

DB'B:  IF     inventory_company_B@company_B_meta_class(slot_desc:T\=dept)
       THEN   new_rule('IF I@inventory_company_B(dept:D,T:Q)
                        THEN inventory(dept:D,item:T,quantity:Q)')
              ⇒ deductive_rule

DB'C:  IF     T@company_C_meta_class(slot_desc⊇[dept,quantity]) and
              prolog{string_concat(T,'_company_C',T1)}
       THEN   new_rule('IF I@T1(dept:D, quantity:Q)
                        THEN inventory(dept:D,item:T,quantity:Q)')
              ⇒ deductive_rule
```

Figure 7. Translation of deductive rules of Figure 6

The heterogeneity of these databases is evident. The concept of items is represented as atomic values in `company_A`, as attributes in `company_B`, and as classes in `company_C`. Without loss of generality, we assume that the item names are the same in each database since it is not difficult to map different names using our deductive rule language.

External Schema References. The first database `company_A` is an external database that shares the same schema with the common view. An external relational or OODB schema is translated into InterBase^{KB} as a collection of classes. The schema of the class is the same as the schema of the corresponding external relation or class, concerning the names and types of attributes. A relation/class is imported in InterBase^{KB} using a deductive rule for defining a derived class as a "mirror" of the external entity. The external (base) class is represented in the condition of the rule using the normal rule syntax extended with a reference to the name of the external database.

The class `inventory` of database `company_A` is imported into InterBase^{KB} as shown in Figure 6. The name of the database from which the relation/class is imported appears just after the name of the class. Each imported database is represented in InterBase^{KB} as a meta-class. This meta-class contains all the necessary information about the imported database, such as its name, type, network address of CSI

and CDB, exported relation/classes, communication and/or storage protocols, etc. This information is copied from the system's data directory [48].

Figure 7 shows the translated deductive rules of **Figure 6**. The translation of rule DB'_A is straight forward because it just contains a reference to an external database and the classes of the external databases are automatically renamed, thus appending the name of the database.

Second-order Syntax. The derived class `inventory` will be also used to import inventory data from the rest of the company databases. However, the import of the other databases cannot be done in such a straightforward manner because items are either attribute or class names, therefore a second order syntax is needed. When variables of a deductive rule language can range over attribute or class names, we say that the rule language has second-order syntax. The databases for `company_B` and `company_C` are imported as shown in Figure 6.

Rule DB_B has a variable `C` that ranges over all the attributes of the class `inventory` of database `company_B`, except attribute `dept`, which is explicitly mentioned in the condition. Rule DB_C has again a variable `C` that ranges over the classes of database `company_C`. Despite the second-order syntax, the above rules are interpreted using a set of first-order rules using the meta-classes of the OODB schema. Each second-order reference for OODB classes in a rule is transformed into a first-order reference to the equivalent OODB meta-classes. Furthermore, a deductive rule that contains second-order syntax is transformed into a production rule that creates first-order deductive rules. Figure 7 shows the translated deductive rules of **Figure 6**.

Concerning the condition, the intra-object (class) pattern is replaced with a pattern of the meta-class of the class. The new meta-class pattern can match its instances, which are the classes to be “discovered” by the second-order constructs of the original rule. The attribute patterns of the original rule are transformed in attribute tests of the `slot_desc` attribute of the meta-class. This set-valued attribute is present in every meta-class and contains the description (name, type, cardinality, visibility, etc.) for each attribute of its instances (classes).

The condition of the second rule that contains a variable for an attribute name ($item_i$) is directly translated into a meta-class pattern whose slot_desc attribute is retrieved and propagated to the rule action. Thus, the variable C stands now for a value of the slot_desc attribute, and the second-order construct is transformed into first-order. Since the class name (inventory_company_B) is known from the original rule, the instance of the meta-class (company_B_meta_class) in the meta-class pattern is instantiated.

The condition of the third rule is slightly different since the class name is a variable in the original rule. Therefore, the variable now appears as the OID of the instance of the meta-class pattern. Furthermore, the selection of the classes among the instances of the meta-class (company_C_meta_class) is restricted to those that have at least the attributes dept and quantity.

The transformed rules are production rules and their action is a method call to create a new deductive rule. The deductive rule has first-order syntax since any variables that appear in the place of attributes or classes have already been instantiated by the condition with the meta-class pattern. Rule DB'_C also contains a call to a Prolog built-in predicate in order to construct the proper name for the $item_i$ classes. Similar calls are also included in the actual implementation for creating the rule strings (e.g. for incorporating the variables), but are omitted here to ease the presentation.

The production rules of Figure 7 are triggered even if they are generated after the creation of the class and meta-class schema because the A-KB core includes a rule activation phase at the end of rule creation. Furthermore, the A-KB core creates events for every method of the OODB schema, including meta-classes. Rules DB'_B, DB'_C will be fired as many times as the number of items in the respective databases and the same number of deductive rules will be generated. The new deductive rules will also be activated and fired based on the same mechanism. Rule DB'_A is a deductive rule and it will behave as described in Section IV.A.

c. Integration transparency. After local schemata have been translated into the common data model and a single global schema exists, the users of the data warehouse are unaware of the origins of data. Instead the system distributes their requests transparently to the appropriate data source. This is achieved using similar mechanisms as those described in section IV.B.2. More details can be found in [47].

V. Conclusions and Future Directions

In this section, we summarize the material we have presented throughout this chapter. We have initially presented how various database systems have been extended with active and production rules turning them into active Database systems. In the sequel, we have presented various techniques that integrate multiple rule types into the same database system resulting in active Knowledge Base systems.

We have, finally, described in detail the DEVICE system, which integrates production rules into an active OODB that supports only the lowest-level ECA rules. Then, we have presented two applications based on the DEVICE system, namely deductive databases and data warehousing. The DEVICE system is quite extensible and allows the implementation of various rule types on top of production rules. The result of such an extension is a flexible, yet efficient, active KBMS that allows the user to work with many rule types, according to the application type and/or his/her programming proficiency level.

More specifically, the highest the rule type level, the more naive programmer the user can be. For example, deductive rules are used for specifying complex views in a declarative manner, which are more or less queries to the data. Production rules are used for programming in an expert system style and enforcing integrity constraints. Finally, ECA rules can be used for a variety of data maintenance, security, and integrity enforcement tasks, including programming applications.

The current trend in knowledge base systems is to develop large-scale knowledge-based applications in order to overcome the difficulties in developing intelligent software. We believe that the next decade will establish knowledge-based applications into the mainstream of software technology

since the demanding complexity of modern real-world problems requires the use of human expertise to be dealt with.

A distinctive feature that knowledge base systems must have for future applications is activeness. Active knowledge base systems will respond intelligently to emerging situations without user intervention. Knowledge-based systems that are built on such a reactive behavior will be able to control complex distributed systems in a seamless manner.

Appendix

<production_rule> ::= if <condition> then <action>

<deductive_rule> ::= if <condition> then <derived_class_template>

<derived_attribute_rule> ::= if <condition> then <derived_attribute_template>

<condition> ::= <inter-object-pattern>

<inter-object-pattern> ::= <condition-element> ['and' <inter-object-pattern>]

<inter-object-pattern> ::= <inter-object-pattern> 'and' <prolog_cond>

<condition-element> ::= ['not'] <intra-object-pattern>

<intra-object-pattern> ::= [<var>'@']<class>['('<attr-patterns>')']

<attr-patterns> ::= <attr-pattern>[','<attr-patterns>]

<attr-pattern> ::= <var-assignment> | <predicate>

<attr-pattern> ::= <attr-function>':'<var> <rel-operator> <value>

<var-assignment> ::= <attr-function>':'<var>

<predicate> ::= <attr-function> <predicates>

<predicates> ::= <rel-operator> <value> [{ & | ; } <predicates>]

<rel-operator> ::= = | > | >= | =< | < | \=

<value> ::= <constant> | <var>

<attr-function> ::= [<attr-function>'.']<attribute>

<prolog_cond> ::= 'prolog' '{'<prolog_goal>'}'

<action> ::= <prolog_goal>

<derived_class_template> ::= <derived_class>'('<templ-patterns>')'

<derived_attribute_template> ::= <var>'@'{<class>}'('<templ-patterns>')'

<templ-patterns> ::= <templ-pattern> [',' <templ-pattern>]

<templ-pattern> ::= <attribute>':'{<value> | <aggr_func>'('<var>')'}

<aggr_func> ::= count | sum | avg | max | min

<class> ::= *An existing OODB class or derived class*

<derived_class> ::= *An existing OODB derived class or a non-existing OODB class*

<attribute> ::= *An existing attribute of the corresponding OODB class or derived class*

<prolog_goal> ::= *An arbitrary Prolog/ADAM goal*

<constant> ::= *A valid constant of an OODB simple attribute type*

<var> ::= *A valid Prolog variable*

References

[1] J. Ullman, *Principles of Database and Knowledge-Base Systems*. Rockville, Maryland: Computer Science Press, 1989.

[2] S. Ceri, G. Gottlob, and L. Tanca, *Logic Programming and Databases*. Berlin: Springer-Verlag, 1990.

[3] J. Minker, *Foundations of Deductive Databases and Logic Programming*, Los Altos: Morgan Kaufmann, 1988.

[4] E.N. Hanson, The Ariel project, in *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom and S. Ceri, Eds.: Morgan Kaufmann Publishers, 1996, pp. 177-206.

[5] J. Widom, Deductive and active databases: Two paradigms or ends of a spectrum?, *Int. Workshop on Rules in Database Systems*, Edinburgh, Scotland, 1993, pp. 306-315.

[6] E.N. Hanson and J. Widom, An overview of production rules in database systems, *The Knowledge Engineering Review*, **8**(2), pp. 121-143, 1993.

[7] C.L. Forgy, OPS5 user manual, Dept. of Computer Science, Carnegie-Mellon University 1981.

[8] C.L. Forgy, RETE: A fast algorithm for the many pattern/many object pattern match problem, *Artificial Intelligence*, **19**, pp. 17-37, 1982.

[9] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra, Design of Sentinel: An object-oriented DBMS with event-based rules, *Information and Software Technology*, **39**(9), pp. 555-568, 1994.

[10] U. Dayal, A.P. Buchman, and D.R. McCarthy, The HiPAC project, in *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom and S. Ceri, Eds.: Morgan Kaufmann Publishers, 1996, pp. 177-206.

[11] O. Diaz, N. Paton, and P.M.D. Gray, Rule management in object oriented databases: A uniform approach, *Int. Conf. on Very Large Databases*, Barcelona, Spain, 1991, pp. 317-326.

[12] S. Gatziu, A. Geppert, and K.R. Dittrich, Integrating active concepts into an object-oriented database system, *Workshop on Database Programming Languages*, Nafplion, Greece, 1991, pp. 399-415.

[13] H. Branding, A.P. Buchmann, T. Kudrass, and J. Zimmermann, Rules in an open system: The REACH rule system, *Int. Workshop on Rules in Database Systems*, Edinburgh, Scotland, 1993, pp. 111-126.

[14] O. Diaz and A. Jaime, EXACT: An extensible approach to active object-oriented databases, Dept. of Languages and Information Systems, University of the Basque Country, San Sebastian, Spain 1994.

[15] T. Risch and M. Skold, Active rules based on object-oriented queries, *IEEE Data Engineering Bulletin*, **15**(4), pp. 27-30, 1992.

[16] M. Berndtsson and B. Lings, On developing reactive object-oriented databases, *IEEE Data Engineering Bulletin*, **15**(4), pp. 31-34, 1992.

[17] C. Collet, T. Coupaye, and T. Svensen, NAOS - Efficient and modular reactive capabilities in an object-oriented database system, *Int. Conf. on Very Large Databases*, Santiago, Chile, 1994, pp. 132-143.

[18] S. Chakravarthy and D. Mishra, Snoop: An expressive event specification language for active databases, *Data & Knowledge Engineering*, **14**(1), pp. 1-26, 1994.

[19] S. Gatziu and K.R. Dittrich, Events in an active object-oriented database, *Int. Workshop on Rules in Database Systems*, Edinburgh, Scotland, 1993, pp. 23-39.

[20] N.H. Gehani, H.V. Jagadish, and O. Shmueli, Event specification in an active object-oriented database, *ACM SIGMOD Int. Conf. on the Management of Data*, 1992, pp. 81-90.

[21] N. Bassiliades and I. Vlahavas, DEVICE: Compiling production rules into event-driven rules using complex events, *Information and Software Technology*, **39**(5), pp. 331-342, 1997.

[22] S. Potamianos and M. Stonebraker, The POSTGRES rule system, in *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom and S. Ceri, Eds.: Morgan Kaufmann Publishers, 1996, pp. 177-206.

[23] J. Widom, The Starburst rule system, in *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom and S. Ceri, Eds.: Morgan Kaufmann Publishers, 1996, pp. 177-206.

[24] E. Simon, J. Kiernan, and C.d. Maindreville, Implementing high level active rules on top of a relational DBMS, *Int. Conf. on Very Large Databases*, Vancouver, Canada, 1992, pp. 315-326.

[25] T.A.-N. Consortium, The active database management system manifesto: A rulebase of ADBMS features, *SIGMOD Record*, **25**(3), pp. 40-49, 1996.

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

[27] L.M.L. Delcambre and J. Etheredge, The Relational Production Language: A production language for relational databases, *Int. Conf. on Expert Database Systems*, Vienna, Virginia, 1988, pp. 333-351.

[28] J. Kiernan, C.d. Maindreville, and E. Simon, Making deductive databases a practical technology: A step forward, *ACM SIGMOD Int. Conf. on the Management of Data*, Atlantic City, NJ, 1990, pp. 237-246.

[29] T. Sellis, C.-C. Lin, and L. Raschild, Coupling production systems and database systems: A homogeneous approach, *IEEE Trans. on Knowledge and Data Engineering*, **5**(2), pp. 240-255, 1993.

[30] D.A. Brant and D.P. Miranker, Index support for rule activation, *ACM SIGMOD Int. Conf. on the Management of Data*, 1993, pp. 42-48.

[31] D.P. Miranker, TREAT: A better match algorithm for AI production systems, *AAAI*, 1987, pp. 42-47.

[32] E.N. Hanson, Gator: A generalized discrimination network for production database rule matching, *IJCAI Workshop on Production Systems and their Innovative Applications*, 1993.

[33] C.d. Maindreville and E. Simon, A production rule based approach to deductive databases, *IEEE Int. Conf. on Data Engineering*, 1988, pp. 234-241.

[34] E. Simon and J. Kiernan, The A-RDL System, in *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widom and S. Ceri, Eds.: Morgan Kaufmann Publishers, 1996, pp. 111-149.

[35] J.V. Harrison and S.W. Dietrich, Integrating active and deductive rules, *Int. Workshop on Rules in Database Systems*, Edinburgh, Scotland, 1993, pp. 288-305.

[36] N.W. Paton, Supporting production rules using ECA rules in an object-oriented context, *Information and Software Technology*, **37**(12), pp. 691-699, 1995.

[37] M. Skold and T. Risch, Using partial differencing for efficient monitoring of deferred complex rule conditions, *IEEE Int. Conf. on Data Engineering*, 1996, pp. 392-401.

[38] S. Ceri and J. Widom, Deriving incremental production rules for deductive data, *Information Systems*, **19**(6), pp. 467-490, 1994.

[39] U. Griefahn and R. Manthey, Update propagation in Chimera, an active DOOD language, *Int. Workshop on the Deductive Approach to Information Systems and Databases*, Spain, 1994, pp. 277-298.

[40] N. Bassiliades and I. Vlahavas, Processing production rules in DEVICE, an active knowledge base system, *Data & Knowledge Engineering*, **24**(2), pp. 117-155, 1997.

[41] N. Bassiliades, I. Vlahavas, and A. Elmagarmid, E-DEVICE: An extensible knowledge base system with multiple rule support, Dept. of Computer Science, Purdue University, W. Lafayette, Indiana, Technical Report, CSD-TR #97-048, October 1997.

[42] N.W. Paton, ADAM: An object-oriented database system implemented in Prolog, *British National Conf. on Databases*, 1989, pp. 147-161.

[43] P.M.D. Gray, K.G. Kulkarni, and N.W. Paton, *Object-Oriented Databases, A Semantic Data Model Approach*. London: Prentice Hall, 1992.

[44] A. Gupta, I.S. Mumick, and V.S. Subrahmanian, Maintaining views incrementally, *ACM SIGMOD Int. Conf. on the Management of Data*, 1993, pp. 157-166.

[45] J. Ullman, A comparison between deductive and object-oriented database systems, *Int. Conf. on Deductive and Object-Oriented Databases*, Munich, 1991, pp. 263-277.

[46] J. Widom, “Special Issue on Materialized Views and Data Warehousing,” in *IEEE Data Engineering Bulletin*, vol. 18(2), D. Lomet, Ed., 1995.

[47] N. Bassiliades, I. Vlahavas, A.K. Elmagarmid, and E.N. Houstis, InterBase^{KB}: A knowledge-based multi-database system for data warehousing, Dept. of Computer Science, Purdue University, W. Lafayette, Indiana, Technical Report, CSD-TR #97-047, October 1997.

[48] J. Mullen, O. Bukhres, and A. Elmagarmid, “InterBase*: A Multi-database System,” Object-Oriented *Multi-database Systems,* O. Bukhres and A. K. Elmagarmid, Eds., Prentice Hall, 1995, pp. 652-683.

[49] J.G. Mullen and A. Elmagarmid, “InterSQL: A Multi-database Transaction Programming Language,” Proc. Workshop on Database Programming Languages, 1993, pp. 399-416.

[50] D. Quass, A. Gupta, I.S. Mumick, and J. Widom, “Making Views Self-Maintainable for Data Warehousing,” Proc. Conf. on Parallel and Distributed Information Systems, Miami, Florida, USA, 1996.

[51] Y. Zhuge, H. Garcia-Mollina, J. Hammer, and J. Widom, “View maintenance in a warehousing environment,” Proc. ACM SIGMOD Int. Conf. on the Management of Data, San Jose, California, 1995, pp. 316-327.

[52] E. Pitoura, O. Bukhres, and A. Elmagarmid, “Object Orientation in Multi-database Systems,” *ACM Computing Surveys*, Vol. 27, No. 2, 1995, pp. 141-195.

[53] W. Kim, I. Choi, S. Gala, and M. Scheevel, “On resolving schematic heterogeneity in multi-database systems,” Distributed and Parallel Databases, Vol. 1, No. 3, 1993, pp. 251-279.

[54] C. Batini, M. Lenzerini, and S.B. Navathe, “Comparison of methodologies for database schema integration,” ACM Computing Surveys, Vol. 18, No. 4, 1986, pp. 323-364.