E-DEVICE: An Extensible Active Knowledge Base System with Multiple Rule Type Support[§]

[†]Nick Bassiliades[*] , [†]Ioannis Vlahavas[**] and [‡]Ahmed K. Elmagarmid[***]

[†]Dept. of Informatics
Aristotle University of Thessaloniki,
54006 Thessaloniki, Greece.
E-mail: {nbassili,vlahavas}@csd.auth.gr

[‡]Dept. of Computer Sciences
Purdue University
West Lafayette, IN 47907-1398, USA
E-mail: ake@cs.purdue.edu

[§] This work is partially supported by EU under the INCO-COPERNICUS project (ref. No. 960114).

[*] Supported by a scholarship from the Greek Foundation of State Scholarships (F.S.S. - I.K.Y.).

[**] This work was carried out while the author was on sabbatical leave at Purdue University.

[***] Partially supported by a grant from the Intel Corporation.

Abstract

This paper describes E-DEVICE, an extensible active knowledge base system (KBS) that supports the processing of event-driven, production, and deductive rules into the same active OODB system. E-DEVICE provides the infrastructure for the smooth integration of various declarative rule types, such as production and deductive rules, into an active OODB system that supports low-level event-driven rules only by a) mapping each declarative rule into one event-driven rule, offering centralized rule selection control for correct run-time behavior and conflict resolution, and b) using complex events to map the conditions of declarative rules and monitor the database to incrementally match those conditions. E-DEVICE provides the infrastructure for easily extending the system by adding a) new rule types as subtypes of existing ones and b) transparent optimizations to the rule matching network. The resulting system is a flexible, yet efficient, KBS that gives the user the ability to express knowledge in a variety of high-level forms for advanced problem solving in data intensive applications.

Keywords: knowledge base system, production rules, deductive rules, derived attributes, aggregation, negation, active object-oriented database

1. Introduction

Knowledge Base Systems (KBS) have emerged as an integration of database and knowledge-based (or rule-based) system technologies [36] which satisfy the demand for intelligent, data-intensive applications. On the other hand, advanced data representation features are required by new complex database applications. It has been proposed that Object-Oriented Databases (OODBs) [19] can be suitable for satisfying the need of both designers and users for a representation tool that is semantically richer than that provided by relational databases.

There are two main areas where research in rule integration is being done: active and deductive databases [38]. Active databases use low-level *situation-action* rules which are triggered when a *situation* arises in the database while deductive databases are composed from high-level rules which declaratively describe data in terms of other data, without an exact description of how new data are treated. Furthermore, active rules take the form of event-driven and data-driven rules. Data-driven or production rules are more declarative than event-driven rules [23] because they describe a firing situation without an exact definition of the way the situation is detected. Henceforth, we will call production and deductive rules collectively as *declarative rules*, due to the declarative nature of their condition. Event-driven or Event-Condition-Action (ECA) rules are more procedural because they explicitly define their activation [38].

All three rule types are useful in an active KBS: event-driven rules are needed mainly for implementing time constrained applications and extensions to the database system; data-driven rules are useful for monitoring database states or enforcing integrity constraints; and deductive rules are useful to define and maintain recursive, materialized views in the database system [8, 21]. Although there are a few relational systems that support two or more rule types, very few OODB systems do so.

In this paper, we present an extensible object-oriented KBS, called E-DEVICE (Extensible - Data-driven & EVent-driven rule Integration using Complex Events), that integrates production and deductive rules into an active OODB that generically supports event-driven rules only. The integration is based on the compilation of the condition of both declarative rule types into a discrimination network that consists of simple and complex events which record and combine database modifications that could possibly make a rule fire.

The paper presents the architecture of the system and the relationship between its components, giving special emphasis to the extensibility of the system. E-DEVICE is a large extension of our previous system DEVICE [3, 4], an expert database system shell that supports production rules only. E-DEVICE supports several new rule types, such as deductive rules, rules for derived and aggregate attributes and provides modularity and extensibility so that new rule types can be added seamlessly. This has been achieved by abstracting the semantics of production rules using truth maintenance techniques and by providing an extensible rule type hierarchy.

The extensibility of the system is due to the use of the object-oriented (OO) infrastructure of metaclasses which allows for partial redefinition of rule managers as well as transparent extensions and improvements of the rule matching and execution phases. In the paper we describe several such extensions, such as new rule managers for integrating materialized and non-materialized deductive rules, derived and aggregate attributes, transparent optimizations to the storage and retrieval of partially matched information in the event network and set-oriented rule execution.

The structure of the paper is as follows: Section 2 refers to related work on rule integration in active OODBs; Section 3 presents the overall system architecture; Section 4 presents the declarative rule language; Section 5 describes the rule integration scheme; Section 6 describes the extensibility and optimization features of E-DEVICE; Section 7 discusses the system performance, presenting results for deductive rules; and finally, section 8 concludes this paper with a summary of the main points and a discussion of future work.

2. Related Work

There are several active OODB systems that support ECA rules [9, 10, 12, 11, 15]. The ECA rule type is the most easy to implement in OODBs since events conform to the message-passing paradigm of OO computation and every recognizable message/method can be a potential event. Therefore, they can be executed as a "detour" from normal method execution. There are a few active OODB systems that support production and/or deductive rules in addition to ECA rules, but there is no system, to the best of our knowledge, that supports both rule types.

ODE [16] supports both ECA rules (triggers) and production rules (constraints), but their use is rather primitive because triggers must be explicitly evoked by member functions and all relevant constraints are inefficiently checked every time an object is accessed. There is no notion of incremental condition checking.

The Chimera prototype [20] and OPRA [31], an extension of the ADAM/EXACT OODB [12, 11], support deductive rules and production rules, respectively (but not both), by translating each high-level rule into a set of ECA rules using the technique first described by Ceri and Widom [8]. This approach is simpler than ours but has certain drawbacks, such as the delay of propagating the effects of object deletion due to re-derivation of previously derived objects. Furthermore, the performance comparison of our approach with [31] (in [4]) showed that, in cases of bulk data loading and value-based joins, E-DEVICE is faster while the one-to-many approach is better at joins through object references.

There are several relational database systems, on the other hand, that support multiple rule types. For example, Ariel [23] supports production rules that are efficiently matched using the special discrimination network A-TREAT, which is built from scratch. ECA rules are emulated using production rules and special differential files (or delta relations) that record data modifications to compensate for the lack of "pure" events in a relational database environment. The same architectural line is followed in the RDL1/A-RDL system [24, 35] which additionally supports deductive rules by treating them as special production rules (see section 5.3).

A rather different approach is followed by U-Datalog [7, 6] where deductive rules are extended with updates; therefore, production rules can be emulated. U-Datalog allows updates to appear in the rule body and be executed only when the body condition is satisfied. The rule head, however, keeps the semantics of Datalog rules. The most interesting property of U-Datalog rules is that they have fully declarative semantics even for the updates which are deferred and only executed at the end of query processing if they do not introduce inconsistencies nor redundancies to the database. The declarative specification of updates allows static rule analysis that helps to detect rule termination and confluence and with optimize rule execution.

In E-DEVICE, we follow a different integration path. The core system is an active OODB that supports events and ECA rules. The integration is based on incrementally matching the conditions of declarative rules with a discrimination network. The latter is built from the primitives of the active OODB (events). A tightly-coupled implementation from scratch (like in Ariel and A-RDL) would require the introduction of new low-level data structures that do not blend well with the OO model. Specifically, rule matching would require passing database updates from the OODB to the discrimination function and the latter would pass back the selected rule instantiations. However, the above information is part of the OODB, therefore the low-level function would either access data by-passing the OO interface which would violate encapsulation and integrity, or it would use the message passing facility of the OODB which makes it no longer low-level.

Re-using the primitives of an extensible active OODB system provides us with all the advantages of the OO technology: a) extensibility for the rule system, b) ready-to-use persistence for the discrimination network in the form of events as first-class objects, c) graceful coexistence with ECA rules, and d) flexibility for experimenting and optimizing separately the various parts of the discrimination network.

DIPS [34] is an example of a relational system that re-uses the primitives of the system (relational tables) (rather than a low-level approach) to perform rule condition matching. However, DIPS does not support traditional ECA rules.

There are also relational systems, like POSTGRES [33] and Starburst [39], that originally supported ECA rules. POSTGRES also supports backward chaining deductive rules through data retrieval events which trigger ECA rules that retrieve the derived data. Starburst has been extended with forward chaining deductive rules using a multiple active rule translation method [8], similar to [20, 31]. Apart from the advantages described in section 5.1, performance tests in section 7 clearly show that E-DEVICE is faster for incremental insertions and deletions and also considerably faster for bulk inserts using set-oriented rule execution.

Our algorithm for the maintenance of derived objects is based on a counting algorithm that was first described in [21]. However, in [21] the counting algorithm is used only for non-recursive rules while for recursive ones an algorithm similar to [8] is used. However, we use the counting algorithm in conjunction with the event-driven mechanism of active database; thus, we are able to treat recursive rules as well since events are raised only once and the derivation process always terminates, even for derived objects with infinitely many derivations.

Finally, the work described in this paper extends significantly our previous work in [3, 4] by generalizing the semantics of production rules into those of declarative rules using truth maintenance techniques. This creates a general framework under which several new rule types can be added seamlessly into the system, such as deductive rules and rules for derived and aggregate attributes. Furthermore, E-DEVICE provides an extensible rule type hierarchy under which all rule types can be organized and properly managed.

Compared to the PARK [17] formal semantics defined for active rules, declarative rules of E-DEVICE are a bit different. Deductive rules and rules for derived and aggregate attributes are equivalent to PARK active rules because their "action" part consists of a single action. However, PARK semantics are defined only for inflationary fixpoints, i.e. only for deriving new objects and not for maintaining derived relations. Furthermore, production rules of PARK consist only of a single action while production rules of E-DEVICE include many actions and allow calls to arbitrary Prolog goals; therefore, the analysis of rules performed by the PARK fixpoint algorithm cannot be applied easily. Finally, the conflict resolution strategies of the various rule types of E-DEVICE are also orthogonal to the rule semantics of E-DEVICE, in much the same way as PARK. However, the difference is that E-DEVICE is based on "translating" conflict resolution strategies to rule priorities, while PARK semantics block execution of conflicting rules.

3. The Architecture of E-DEVICE

E-DEVICE is implemented on top of the ECLiPSe Prolog as an extension to the active OODB EXACT [11], which is itself an extension of ADAM OODB [30]. The overall architecture of the E-DEVICE system is shown in Figure 1. The various components are plugged-in as modules, extending the basic active OODB system, rather than being placed on a distinct layer on top of the EXACT and ADAM systems. This is a consequence of the OODB extensibility through metaclasses [32] and makes DEVICE itself extensible. The object-oriented model and terminology we use throughout the paper is based on the ADAM OODB [30].

Figure 1 shows the architecture and operation of E-DEVICE. The user interacts with the system issuing ADAM commands, which are actually Prolog goals extended with message passing. Declarative rules inserted by the user are parsed, pre-compiled, optimized and compiled down to ECA rules. After rules and the corresponding triggering events are created, the database is monitored for updates that possibly make a rule fire. An event manager propagates the events/updates through a discrimination network and informs the corresponding rule manager when rules can be executed. The execution of rules causes database updates that may trigger other rules, and so on so forth.

The compile-time components of the E-DEVICE include a) the parser, which parses the textual description of the rule and, furthermore, applies a series of transformation to the parse tree in order to produce a more efficient and easy to compile tree (pre-compiler/optimizer) and b) the compiler which compiles the parse tree into a network of complex events and an active rule using the techniques described in section 5.

The run-time components of E-DEVICE are the OODB metaclasses that were introduced in EXACT/ADAM in order to implement the structure and behavior of production, deductive, etc. rules (see section 5). Rules are first-class objects, instances of the rule managers. The ECA rule manager is the most generic rule manager and is part of the EXACT system. The various rule managers that implement the functionality of E-DEVICE are subclasses of the ECA rule manager which means that they inherit part of the functions of the generic rule manager while they re-define many of them in order to capture the higher-level semantics of production and deductive rules.

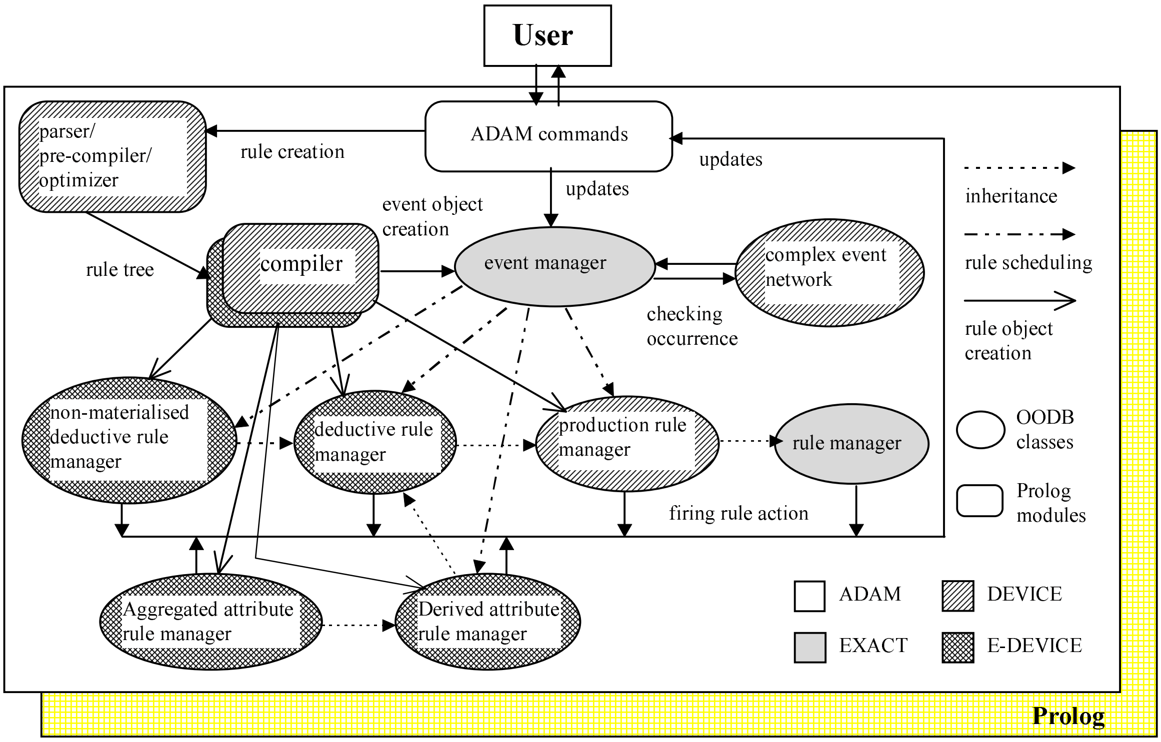

Figure 1. The architecture of the E-DEVICE system.

In addition, new types of events (complex events) have been introduced as subtypes of the generic EXACT events. These are fixed, and they are the building components of the complex event discrimination network that is used to match the declarative rules' conditions. The event manager keeps track of which simple events have occurred and combines them incrementally to match the rules' conditions.

During the development of E-DEVICE, we tried not to interfere and alter the original code for EXACT in order to make E-DEVICE portable across extensible active OODB systems. However, certain features of E-DEVICE were difficult to be implemented without altering the structure and behavior of some EXACT objects. In the future, we will try to optimize the E-DEVICE implementation; one of our goals is to provide an even smoother integration into EXACT.

4. The Declarative Rule Language

This section describes the system's declarative rule language which follows, for the most part, the OPS5 [13] paradigm influenced by the OODB context of E-DEVICE. Both types of declarative rules which are supported by E-DEVICE are expressed as a *condition*, which defines a pattern of objects to be detected over the database, followed by an *action* (for production rules) or a *conclusion* (for deductive rules). The language for defining low-level ECA rules is described in [12, 11].

4.1 The Rule Condition

The following rule condition defines an employee named 'Mike' whose salary exceeds his manager's salary:

Example 1.

```
IF    E@emp(name='Mike',salary:S,manager:M) and M@emp(salary<S)
THEN  delete => E
```

[][1]

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. The first of the above intra-object patterns denotes an instance `E` of class `emp` with attribute `name` equal to `Mike`, with salary `S` and manager `M`. The second intra-object pattern describes the manager `M` of employee `E`, whose `salary` attribute is less than the salary `S` 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 Example 1 could be also written as:

```
E@emp(name='Mike',salary:S,salary.manager<S)
```

Attribute patterns can navigate through object references of complex attributes, such as the complex attribute `salary.manager`. 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. 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 Example 1.

Notice that we have adopted a right-to-left order of attributes, contrary to the C-like dot notation that is commonly assumed, because we would like to stress out the functional data model origins of ADAM OODB [19]. Under this interpretation the chained "dotted" attributes can be seen as function compositions.

Negative conditions are also allowed. A negative condition is satisfied when no objects in the database satisfy the corresponding positive condition. The rule condition of example 2 identifies an employee named 'Mike' whose salary is the highest.

We notice that only safe rules [36] are allowed, i.e. a) variables that appear in the action must also appear at least once inside a non-negated condition and b) variables that are used inside a negated condition must also appear at least once inside a non-negated condition. Otherwise they are just ignored.

---

[1] This symbol marks the end of the example.

Example 2.

```
    IF    E1@emp(name='Mike',salary:S) and
          not E2@emp(salary>S) and
           prolog{NS is S * 0.8}
     THEN  update_salary([S,NS]) ⇒ E1
```

[]

The choice for the logic-like condition language is justified by the fact that the condition is supposed to be a declarative specification of the state of the database, and therefore, it is not appropriate to use the procedural interface of the OODB as the condition language. However, 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 appendix A, we include the full syntax of the condition-part language.

4.2 The Actions of Production Rules

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 [19]. Examples of production rule actions were given in the previous section. In Example 1, the employee named 'Mike' is deleted when his salary is higher than his manager's salary whereas, in Example 2, Mike's salary is lowered by 20% when no other employee has a higher salary.

4.3 The Conclusion of Deductive Rules

Deductive rules have a *conclusion* instead of an action. The conclusion is a derived class template that defines the objects that are derivable when the condition is true.

Example 3.

```
DR1:   IF     A@arc(start:X,end:Y) THEN path(start:X,end:Y)
DR2:   IF     P@path(start:X,end:Y) and A@arc(start:Y,end:Z\=X)
       THEN   path(start:X,end:Z)
```

[]

The deductive rule `DR1` of Example 3 defines that an object with attributes `start`, `end` is an instance of class `path` if there is an object `A` in class `arc` with exactly the same attributes.

Class `path` is a derived class, i.e. a class whose instances are derived from deductive rules. Only one derived class template is allowed at the THEN part (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. For example, the transitive closure of the `arc` relation is completed with the recursive rule `DR2` of Example 3. Objects of derived classes are materialized or not depending on the type of deductive rule. The OID of the derived objects does not appear at the head of the deductive rule because it is not used elsewhere in the rule.

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 appendix A.

5. Declarative Rule Integration

The main idea behind the integration of declarative rules into an ECA based active OODB is that declarative rules are in the form: `IF condition THEN` *`consequent`*, while ECA rules are in the form: `ON event IF condition THEN action`. According to [38], there exist ways to translate the former to the latter or, in other words, to embed high-level rules into an active database system using the primitives of the latter.

Production, deductive, and other declarative rules are compiled to ECA rules in order to be constantly monitored by the active database. The condition of a rule is compiled into a complex event network, which is associated with the event-part of the ECA rule, while the action-part of the ECA rule depends on the type of the declarative rule.

5.1 The Rule Translation Method

The rule translation method of E-DEVICE uses complex events to translate a declarative rule into only one ECA rule. For example consider the following (abstract) rule:

`P1: IF a & b & c THEN <consequent>`

The above production rule is translated into the following ECA rule:

`CA1: ON` e_a `&` e_b `&` e_c `[IF true] THEN <consequent>`

where primitive events e_a, e_b, e_c detect the insertion of the data items `a`, `b`, `c`, respectively, and the operator `&` denotes the conjunction of the events. The complex event manager incrementally detects the occurrence of the complex event by monitoring the primitive events and combining their parameters. The detection of the complex event is equivalent to matching the rule condition. The event manager forwards all parameters to the rule manager which is responsible to execute it.

Notice that when a primitive event occurrence is detected (e.g. e_a), then its parameters must be matched against the parameters of <u>all</u> previously detected event occurrences for the other two events, rather than only with the currently occurred ones. In order to achieve this, the parameters of <u>all</u> 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 condition part of the ECA rule is always `true` because all conditions tests have been incorporated into the complex event. However, some small static conditions are allowed to be checked in the condition part of the ECA rule through the `prolog{}` construct.

An alternative rule integration scheme has been proposed for deductive [8, 20] and production rules [31]. According to this, each high-level rule is translated into many ECA rules, each of which is triggered by a different event, derived from a single condition element of the condition of the high-level rule. The condition of each ECA rule is almost the same as the condition of the high-level rule, minus the event. For example, rule `P1` is translated into the following 3 ECA rules:

A1: ON e_a IF b & c THEN <*consequent*>

A2: ON e_b IF a & c THEN <*consequent*>

A3: ON e_c IF a & b THEN <*consequent*>

In the case of deductive rules, the monitoring of the deletion of objects is also required to keep the database consistent. If the events d_a, d_b, d_c, monitor the deletion of the corresponding data items, then 3 more ECA rules are needed [8, 20]. Furthermore, the approach of [8] requires one more rule to check and re-derive some deleted derived objects due to possible alternative derivations while in [20], deletion is prevented by incorporating this check in the "deletion" rules. E-DEVICE, on the other hand, does not require such a check or a re-derivation because it uses a counter-based mechanism [21] which accounts for the multiply derived objects.

In the following, we analyze the disadvantages of the one-to-many rule translation approach.

Maintenance. If someone wants to delete or temporarily disable a production rule, he/she should 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. Our 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. Consider the following sequence of event occurrences in an empty database: e_c, e_b, e_a. We assume that all events occur in the same transaction and that ECA rules `A1`-`A3` have immediate Event-Condition (EC) coupling mode. ECA rules are considered in the following order: `A3`, `A2`, `A1`. First `A3` and then `A2` are triggered but not executed since their conditions are not satisfied. Finally `A1` is triggered, its condition is satisfied, and the action is executed. This behavior is correct since the production rule `P1` should be fired after the insertion of `c`, `b`, `a`. However, 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 poorer performance, as will be shown in section 8.

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.

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 firing, there is no way to "undo" the rule activation by reversing the effect of the triggering event. 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.

One way to overcome the absence of net effects, in the case of immediate EC and deferred CA coupling modes, is to re-check the condition inside the action of the ECA rule in order to assure 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 `A1` would look under this scheme as follows[2]:

```
A1: ON e_a IF b & c THEN (a&b&c -> <consequent>; true)
```

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

```
A1: ON e_a IF a & b & c THEN <consequent>
```

However, the above solution would incur overhead on the performance of rule execution because it duplicates checking of already checked conditions. Our approach avoids this problem of net event effects as discussed in the next section.

5.2 Production Rule Semantics

The production rule manager receives all the detected complex event occurrences from the event manager and selects those events that activate production rules. The latter, also called “rule instantiations,” are placed into the “conflict set”. On the other hand, the event manager may detect that a previously raised complex event is no longer valid. In this case a negative rule instantiation causes the corresponding positive rule instantiation to be removed from the conflict set, if it still exists 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 be executed [13]. The active database systems approach uses priorities to resolve the rule execution order. In E-DEVICE, we incorporated the OPS5 conflict resolution heuristics 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`.

---

[2] The notation `(a→b;c)` is the notation of Prolog for the usual *if-then-else* programming construct and it means "if `a` (is true) then (execute) `b` else (execute) `c`".

The conflict set is a Prolog list (LIFO structure) that is stored as a class attribute in the production rule manager. The *refractoriness* criteria removes the instantiation tokens of the rules that have been executed from the conflict list. For example, if the following rule exists, refractoriness prevents this rule from being executed ad infinitum for the same employee.

```
IF   E@emp(name:N,age>30)
THEN write(N), write(' is over 30 years old.'), nl
```

The *recency* criteria inserts the newly derived rule instantiations at the beginning of the conflict list, in order to be considered before the older ones. For example, the following two rules are both selectable for execution if there is an employee named 'Mike' in the database with salary less than 30. Let's assume that the first one is selected for execution. A new employee named 'John' will be created. Now, two instantiations of the second rule are selectable. However, the instantiation that concerns 'John' will be executed first because it is more recent than the one of 'Mike'.

```
IF   E1@emp(name='Mike', salary<30)
THEN new([_,[name(['John']),salary([25])]]) ⇒ emp
IF   E1@emp(name:N, salary:S<30) and
     prolog{S1 is 1.1*S}
THEN update_salary([S,S1]) ⇒ E1
```

Finally, the *specificity* criteria selectively picks-up at run-time from the conflict set instantiation tokens of rules 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. In the previous example, the first rule is more specific than the second because its condition checks for a specifically named employee; thus, when both rules are selectable the first one is preferred. This is reflected in the complex event network whereas the network of the first rule has one more logical event object than the network of the second rule (see section 5.4.1).

Notice that because the specificity of a rule is based on the number of actually generated events and <u>not</u> on the syntactical complexity of its condition, deep path expressions in the condition may produce a high specificity score that is at first not comprehensible from the simple high-level rule declaration. For example, the first of the following two rules seems to involve one instance; thus, one might believe that the second rule that involves two instances is more specific.

```
IF   E@emp(name='Mike',salary:S,salary.manager<S)
THEN delete ⇒ E
IF   E1@emp(name='Mike',salary:S) and
     E2@emp(salary<S)
THEN delete ⇒ E
```

However, the first rule is analyzed in the following one during the pre-compilation phase and it is clearly more complex (and specific) than the first one. In fact, the first rule generates 4 complex event objects while the second one only 3.

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

After the rule manager selects a rule instantiation for firing, the condition part of the rule is checked. Usually the trivial `true` condition is associated with E-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 in "immediate" Condition-Action coupling mode.

In E-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 [11]. 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.

5.3 Deductive Rule Semantics

The integration of deductive rules in E-DEVICE is achieved by mapping the deductive rule semantics on production rules. In this section, we describe the materialized approach in which each derived object is stored in the database for future use. We also describe, in the next section, a non-materialized approach where derived objects are not stored but instead computed from the normal (base) objects on demand. Finally, two more types of rules for derived data, namely derived and aggregate attributes, are presented.

According to the semantics of deductive rules, when the condition is satisfied, then the object described in the head exists in the database. This is a declarative way to state that when the condition is satisfied, then the derived object should be inserted in the database. For example, consider the following pair of deductive rules that defines the transitive closure of a network of the path relation between nodes of a graph.

```
IF   A@arc(start:X,finish:Y)
THEN path(start:X,finish:Y)
```

```
IF   P@path(start:X,finish:Z) and
     A@arc(start:Z,finish:Y)
THEN path(start:X,finish:Y)
```

When a new arc object is created then the first rule is executed and a new path object is also created. According to this procedural interpretation, deductive rules can be emulated using production rules, as in RDL1 [24].

However, the simple production rule translation scheme must be extended to fully capture the semantics of deductive rules. For example, 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 the OID-based OODBs and the value-based deductive databases [37].

For example, consider that the following arcs exist in the database: {(1,2), (2,3), (1,3)}. Path (1,3) is directly derivable from the first rule and indirectly derivable from the second rule and the rest of the arcs and paths. The second time this path is derived a new object should not be created.

Furthermore, when the condition of a deductive rule is false, then the derived object of the head must be removed from the database. Before this is done, however, it must be ensured that the derived object is not deducible by another rule instantiation. For example, if arc (1,3) is removed from the database, path (1,3) should not be removed because it is still deducible from the rest of the paths and arcs.

In order to maintain derived classes correctly, we use a counter mechanism which stores the number of derivations of an object [21]. If the derived object has a counter equal to 1, then it is deleted; otherwise the counter is only decreased by 1. In the previous example, the path (1,3) object would have a counter equal to 2 because it can be derived by two distinct ways. Thus, when arc (1,3) is deleted the first rule would just decrease the counter to 1.

Notice here that we use the counting algorithm both for non-recursive and recursive deductive rules while in [21], it is only used for non-recursive rules. This is done in case there are base classes in the rule condition with data cycles among their instances. Then the derived objects have infinitely many derivations and the derivation process will not terminate [29]. For example, if arcs {(1,2), (2,3), (3,2)} are present in the database, then path (1,3) has infinitely many derivations, causing the counting algorithm not to terminate.

Our algorithm, however, terminates in case of infinitely many derivations. When an object is derived more than once, then just its counter is increased. The associated events, which are always raised upon the insertion of the object, are simply not raised because the object is not re-inserted. Therefore, the process terminates because no more events are forwarded into the discrimination network.

Notice that the termination problem discussed above does not have anything to do with the termination problem caused by using functions in the rule language [36]. The language of E-DEVICE is function free, although in section 4.1 we discussed path expressions as function compositions. These expressions are only allowed at the condition of the rule (not at the head) and they are analyzed into function free expressions.

The above operational semantics of deductive rules can be modeled by the following production rules:

```
R1:  IF condition
     THEN (exists(object) → inc_counter(object); create(object))
R2:  IF exists(object) and not(condition)
     THEN (counter(object)>1 → dec_counter(object); delete(object))
```

However, the presence of multiple production rules would yield the system inefficient since the management (at run-time) of both of them would incur unnecessary production cycles.

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 production rule, and the negative action is mapped to the `anti_action` part:

```
IF    condition
THEN  (exists(object) → inc_counter(object); create(object))
ELSE  (counter(object)>1 → dec_counter(object); delete(object))
```

Furthermore, the rule manager is 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, as described in this section.

As it can be concluded from the above discussion, the treatment of deductive rules is quite different from simple production rules. Deductive rules employ a truth maintenance mechanism to maintain the integrity of derived classes when base objects are updated, while production rules do not recover their actions when their condition is no longer satisfied. This behavior is achieved by generalizing the semantics of production rules with the anti-action part.

Derived classes appear at the head of deductive rules. When a deductive rule is created and the derived class that its head refers to does not exist, it is also created. The structure of the class is derived by the template of the head and is extended by the counter we described above. Derived classes allow only retrieval operations on their instances since explicit creation, deletion or modification of derived objects might violate their integrity. Special methods that retrieve instances of derived classes according to their attribute values (like Prolog queries), are provided as well.

The conflict resolution strategies of deductive rules differ from production rules. The recency strategy is set to off and new rule instantiations are appended to the conflict set. The rule search space is navigated in a breadth-first or iterated strategy to model the set-oriented semi-naive evaluation of deductive rules [36].

The specificity criterion is overridden by a new control strategy that considers *stratification* of rules with negation [36]. For example, consider the following pair of rules. Derived class `p` depends negatively upon derived class `q`; thus, in order to evaluate correctly class `p` <u>all</u> objects of class `q` must be derived <u>first</u>. This is called a *stratified* program and it holds only when class `q` does not depend at all on class `p`. The following program has this property, assuming that class `a` is a base class.

```
IF   T@t(attr1:X,attr2:Y) and         IF   A@a(attr6:X)
     not Q@q(attr3:Y)                 THEN q(attr3:X)
THEN p(attr4:X,attr5:Y)
```

When a deductive rule is created, the derived classes that appear in the condition are collected along with their stratum numbers (i.e. their order of evaluation). The algorithm of [36] checks if the new rule, along with the existing ones, constitute a stratified logic program and modifies their strata as a side-effect. The stratum numbers define a partial ordering of rules and is used to resolve rule selection at run-time.

5.3.1 Non-Materialized Deductive Rules

We now consider the non-materialized case, where derived data is not permanently stored in the database, but are computed only as needed as a response to given queries. Non-materialized deductive rules are a special case of the materialized ones and are compiled normally but not activated. Thus the associated primitive events are neither detected nor propagated in the discrimination network.

When a query about a derived class is made, two active rules, signaled before and after the query, are executed. The first rule gathers all the non-materialized derived classes from the dependency graph that contribute to the derivation of the derived class in question and activates all their associated rules. This causes the associated primitive events to be detected retrospectively and propagated into the discrimination network. Finally, the selected rules are executed using the default mechanisms for derived rules and temporarily materialize the derived objects. After the query is executed, the second active rule de-activates all the rules and deletes the derived objects.

Certain optimizations of the non-materialized approach, such as magic sets [36], can be introduced to limit the number of derived objects to just the necessary to answer to a given query. Magic sets rewrite the deductive rules (using additional auxiliary predicates), therefore one possible approach to implement magic sets is to rewrite and compile rules at run-time and then run the query. This approach, of course, has the overhead of compilation and can be used only for large databases and computations. An alternative solution would be to precompile parameterized transformed rule sets using all the alternative goal adornments.

5.3.2 Derived Attributes

E-DEVICE also supports derived attributes, i.e. attributes that are calculated using other attributes of the same or different object(s). Here we describe the materialized approach, i.e. the values for derived attributes are stored and not calculated on-demand. The main concern about the materialized approach is that these values should be maintained when the values of the attributes that they depend upon change.

A derived attribute rule looks like the following:

<u>Example 4.</u>

```
IF    E@emp(salary:S,bonus_percentage:P) and
      prolog{Total is S*(1+P)}
THEN  E@emp(total_income:Total)
```

[]

The above declarative definition of the derived attribute `total_income` implies that:

- when both attributes `salary` and `bonus_percentage` are present, then the derived attribute is calculated and stored.
- when either of the attributes is deleted then the derived attribute must be deleted as well.

Notice that an update of one of the *base* attributes is emulated by a deletion of the attribute, followed by an insertion. Therefore, the above semantics of the derived attribute rules can tackle attribute updates as well. The semantics we describe for the derived attribute rule are emulated through an action/anti-action production rule, in much the same way as the deductive rules presented earlier in the section.

```
IF    E@emp(salary:S,bonus_percentage:P) and
      prolog{Total is S*(1+P)}
THEN  put_total_income([Total]) ⇒ E
ELSE  delete_total_income([Total]) ⇒ E
```

The derived attribute rule manager is a subclass of the deductive rule manager, in order to inherit its run-time behavior (see section 6). However, the two rule managers do not share their conflict sets. Deductive rules are executed using stratification while derived attribute rules are more like production rules but with not any preferred order of execution. Furthermore, the compilation process of derived attribute rules is simpler than deductive rules. The compilation procedure is obvious from the above example.

Apart from the maintenance of the derived attribute, the derived attribute rule also causes the insertion of a new attribute in the class `emp` upon the rule creation. Derived attributes need not be declared when the class is created, but can be inserted later, since the ADAM OODB supports the dynamic re-configuration of the schema.

5.3.3 Aggregate Attributes

In this section, we present how aggregate derived attributes are implemented in E-DEVICE, based on the syntax for derived attributes we described in the previous section.

<u>Example 5.</u>

```
IF    E@emp(salary:S,dept:D)
THEN  D@dept(total_salaries:sum(S))
```

[]

The above rule stores in the derived attribute `total_salaries` of class `dept` the total salary of all employees of that department. In order to correctly maintain the aggregate attribute, the following semantics should be emulated:

- when a new employee is created or his/her salary is inserted or his/her department is inserted, then the salary should be added to the value of the aggregate attribute.
- when an employee is deleted or his/her salary is deleted or his/her department is deleted, then the salary should be subtracted from the value of the aggregate attribute.

```
IF    E@emp(salary:S,dept:D)
THEN  (get_total_salaries(Prev) ⇒ D →
        (Next is Prev + S,
         update_total_salaries([Prev,Next]) ⇒ D);
         put_total_salaries([S]) ⇒ D)
ELSE  (get_total_salaries(Prev) ⇒ D,
       Next is Prev - S,
       update_total_salaries([Prev,Next]) ⇒ D)
--------------------------------------------------------------
IF    E@emp(salary:S,dept:D) and
      AF@aggregate_function(name=sum)
THEN  (get_total_salaries(Prev) ⇒ D →
        (positive_next([Prev,V],Next) ⇒ AF,
         update_total_salaries([Prev,Next]) ⇒ D);
        (get_initial_value(Initial) ⇒ AF,
         positive_next([Initial,V],Next) ⇒ AF,
         put_total_salaries([Next]) ⇒ D))
ELSE  (get_total_salaries(Prev) ⇒ D,
       negative_next([Prev,V,emp,total_salaries],Next) ⇒ AF,
       update_total_salaries([Prev,Next]) ⇒ O)
```

Figure 2. A translated rule for the `sum` aggregate function

Again, the update of the salary or the department is emulated by a deletion followed by an insertion. The semantics we described for the aggregate attribute rule are emulated almost in the same way with derived attribute rules. The translation for the above aggregate attribute rule is given in Figure 2, where both a simplified and the actual translations are shown.

Notice that before the action updates the value of the aggregate attribute, it checks if this is the first time the rule runs. An implied assumption is that the `sum` aggregate function has an initial value of 0. This is a generic description of the semantics of aggregate attribute rules. In reality, the structure of the translated rule is a little more complicated in order to cater for other aggregate functions as well. Furthermore, it is preferable to have an extensible mechanism for aggregate functions, i.e. to be relatively easy to add new ones.

In E-DEVICE, the extensibility is provided by a class called `aggregate_function`, whose instances hold information about the aggregate functions that are supported by the system. This includes its name, initial value, the methods for calculating its value incrementally, and possibly a method for calculating the answer to a query regarding the aggregate attribute (Table 1). The action/anti-action of the transformed rule must access the attributes and methods of the instances of this class in order to able to provide one general translation rule for all kinds of aggregate

class	**aggregate_function**
slots	name initial_value
methods	positive_next negative_next calc_result

Table 1. The class `aggregate_function`

functions. For example, Table 2 shows how the `sum` aggregate function is represented as an instance of the class `aggregate_function`.

The choice of using objects to represent aggregate functions is justified by the fact that the data model and data definition language are object oriented. However, these functions are not built directly into the data definition language because the latter is quite low-level; basically Prolog extended with message passing. Instead we use a more extensible approach where the application programmer is able to add more aggregate functions by providing the appropriate low-level functionality (using the data definition language) and wrapping around it a function object.

The semantics of all but the last method are obvious. To demonstrate the need for the `calc_result` method, recall that the average (`avg`) aggregate function can only be computed by keeping both the sum and the count for the attribute to be averaged. Therefore, the value that is actually stored in the aggregated attribute is not the average value but rather a tuple of the sum and the count. Then the average value can be calculated on demand when the object that contains the aggregate attribute is queried. Then the `calc_result` method is used to calculate the average from the sum and count aggregate values (Table 3). On the other hand, the `sum` aggregate function does not need such a treatment, and the `calc_result` method (Table 2) just returns the stored value. Using this scheme, more advanced

object	**1#aggregate_function**
instance_of	aggregate_function
name	sum
initial_value	0
positive_next([Old,Current],Next) :- Next is Old + Current.	
negative_next([Old,Current,_,_],Next) :- Next is Old - Current.	
calc_result(Sum,Sum).	

Table 2. The aggregate function `sum`

object	**5#aggregate_function**
instance_of	aggregate_function
name	avg
initial_value	[0,0]
positive_next([[OldS,OldC],Current],[NextS,NextC]) :- NextS is OldS + Current, NextC is OldC + 1.	
negative_next([[OldS,OldC],Current,_,_],[NextS,NextC]) :- NextS is OldS - Current, NextC is OldC - 1.	
calc_result([Sum,Count],Avg) :- (Count \= 0 -> Avg is Sum/Count; Avg = 0).	

Table 3. The aggregate function `avg`

object	**3#aggregate_function**
instance_of	aggregate_function
name	max
initial_value	*The smallest value in the domain of the attribute*
positive_next([Old,Current],Next) :- (Current > Old -> Next = Current; Next = Old).	
negative_next([Old,Current,Class,Attrib],Next) :- (Current < Old -> Next = Old; find_max([Attrib],Next) => Class).	
calc_result(Max,Max).	

Table 4. The aggregate function `max`

algebraic aggregate functions [18] that are pretty much useful for data warehousing and OLAP [28] can be incorporated.

Finally, certain aggregate functions, like `max` and `min`, cannot be maintained incrementally upon deletion [28]. For these functions, the `negative_next` method is complicated and inefficient. In Table 4, the method `find_max` finds the maximum value for a certain attribute of a class when the value that has been deleted matches with the current maximum value. Notice that the input argument list of the `negative_next` method is longer than the one of `positive_next` method in order to pass such information like the class and the attribute to be aggregated. These extra argument are only used for aggregate functions like `max` and `min` while for the rest of the functions they are simply ignored (Table 2, Table 3).

Currently, the code for the three methods that are necessary to maintain (and query) aggregate attributes must be manually supplied by the maintainer of the database. However, we believe that it is quite clear where and how these methods are inserted.

In the following, a general procedure for translating an aggregate attribute rule is presented. Assume that the rule is given as:

```
IF    condition(class,O,attrib,V)
THEN  O@aggr_class(aggr_attrib:aggr_func(V))
```

```
IF    condition(class,O,attrib,V) and
      AF@aggregate_function(name=aggr_func)
THEN  (get_<aggr_attrib>(Prev) ⇒ O →
            (positive_next([Prev,V],Next) ⇒ AF,
             update_<aggr_attrib>([Prev,Next]) ⇒ O);
            (get_initial_value(Initial) ⇒ AF,
             positive_next([Initial,V],Next) ⇒ AF,
             put_<aggr_attrib>([Next]) ⇒ O))
ELSE  (get_<aggr_attrib>(Prev) ⇒ O,
       negative_next([Prev,V,class,attrib],Next) ⇒ AF,
       update_<aggr_attrib>([Prev,Next]) ⇒ O)
```

Figure 3. The general form of a translated aggregate attribute rule

where aggr_attrib is the name of the aggregate attribute for class aggr_class, aggr_func is the name of the aggregate function, O is an object of class aggr_class that holds the aggregate attribute value, and V is the value of the attribute attrib of class class to be aggregated. The translated rule is shown in Figure 3. The most notable feature is that the condition has been extended with an intra-object pattern that retrieves the OID of the aggregate function instance, in order to use it for accessing the initial value and the incremental maintenance methods. Furthermore, the anti-action does not need to use the initial value because when it is executed due to a negative rule instantiation the rule must have been positively executed in the past. A more concrete example of translating an aggregate attribute rule is shown in Figure 2.

The aggregated attribute rule manager is a subclass of the derived attribute rule manager and inherits its run-time behavior. Furthermore, the two rule managers share their conflict sets (section 6). However, the compilation process of aggregated attribute rules overrides that of derived attribute rules because it is much more complex.

5.4 Condition Compilation and Matching

The efficient matching of production rules is usually achieved through a discrimination network, like RETE [14], TREAT [25], etc. E-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 active database system. This section overviews both the structure and behavior of the network along with the compilation process that produces it. More details about both the compilation and run-time aspects of the network nodes can be found in [3, 4].

5.4.1 Structure and Behavior of the Complex Event Network

This section describes the structure and behavior of the complex event network that incrementally matches the rule conditions against the database. Throughout this section, we use the following deductive rule. The complex event network for this rule is shown in Figure 4.

```
IF    P@path(start:thessaloniki,end:Y) and
      A@arc(start:Y,end:Z\=thessaloniki)
THEN  path(start:thessaloniki,end:Z)
```

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 4, there are several primitive events, such as put_start, put_end for both classes arc and path, and their delete_ type counterparts.

The fact that the DEVICE network has multiple input sources makes it suitable for a database environment where updates can be asynchronous. Furthermore the fine-grained input sources can capture updates on single object attributes rather than whole database tuples, as in relational databases [23].

The creation of whole database objects could be monitored by the event `new` of the class of the intra-object patterns. However, method `new` iterates over the attributes of the newly created object and inserts them using the default `put_` type methods. Hence, the monitoring of primitive `put_` type events suffice and is more flexible since it can capture simple attribute updates. Similar arguments also hold for the deletion of objects and the corresponding `delete_` type events. Actually, this is an optimization of the compilation that is specific only to EXACT/ADAM. Other extensible OODB systems might require also the monitoring of the object creation event `new`.

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, we introduce *anti-signaling*. 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 nodes via the corresponding `checking` (`anti_checking`) method. These methods internally process the input tokens and check if a complex event can be signaled according to the current input signal and the local history of event activation. When appropriate, these methods construct output tokens that are forwarded further in the event network.

Logical events perform simple attribute tests, and they are only raised when the associated condition is satisfied. In E-DEVICE, logical events map attribute comparisons with constants, and they are signaled by primitive events to

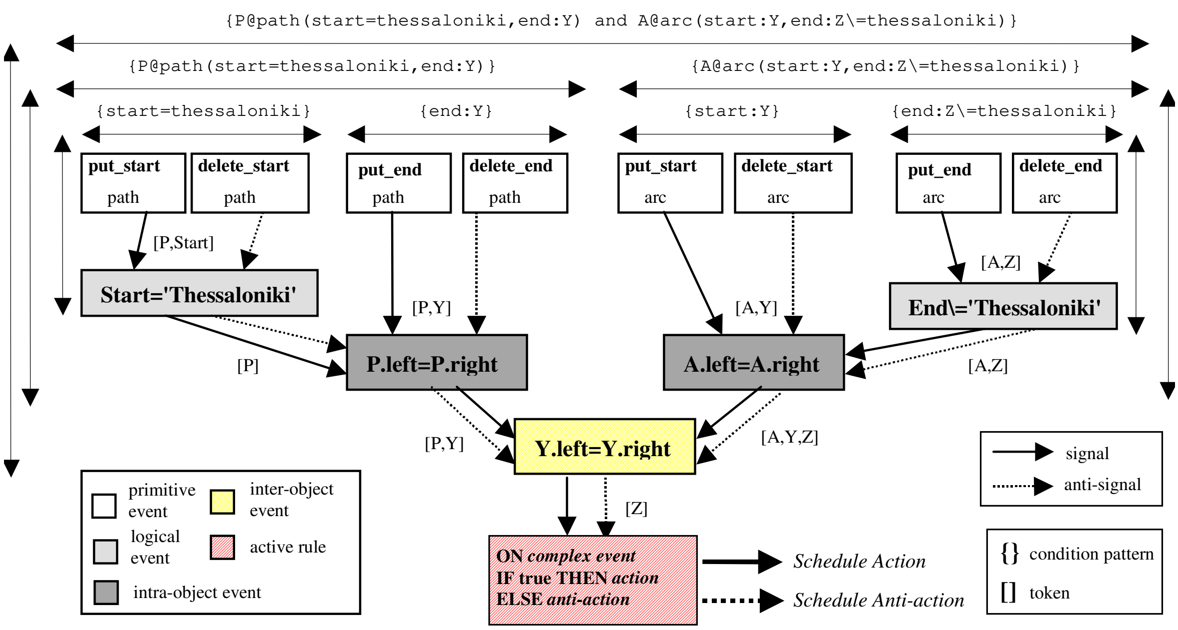

Figure 4. A sample complex event network

perform a check on their parameter. If the check is successful, an output token is propagated to a successor event in the event network. In Figure 4, two such logical events exist, each one of them for the attribute tests against a constant.

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 OID the message recipient objects. In Figure 4, there are two intra-object events since there are two intra-object patterns in the condition of the example rule.

When an intra-object pattern consists of more than two attribute patterns, then the intra-object event that joins the first two attribute patterns is further joined with the remaining attribute patterns into new intra-object events and so on and so forth 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 4, there is one inter-object event that joins the two intra-object events on the value of variable `Y`.

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

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 one or more positive output tokens according to a pre-compiled pattern and are propagated further to the event network.

Example 6.

In Figure 4, assume that there exist 2 arcs that start from the node `'Thessaloniki'` in the database. Both of them will cause the creation of two `path` objects, due to the first rule of Example 9. These two `path` objects correspond to tokens that reside at the left memory of the two-input inter-object event, e.g. `[27#path,'Katerini']` and `[33#path,'Kilkis']`. The name of the starting node is not stored in the tokens because the condition pattern merely tests it. Therefore, it is not forwarded beyond the corresponding logical event. The two original `arc` objects are also stored as tokens in the right memory of the inter-object event: `[12#arc,'Thessaloniki','Katerini']`, `[17#arc,'Thessaloniki','Kilkis']`. If a new `arc` object is inserted in the database, then the token, e.g. `[41#arc,'Kilkis','Serres']` is signaled at the right input, stored at the right memory, and joined with the above two tokens of the left memory. The join succeeds only for the second token; therefore, the token `['Serres']` is signaled at the output of the two-input event.

[]

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 from the event. 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. This deletion optimization [22] is valid because the tokens that flow among two-input events contain the unique OIDs of the objects involved in the condition. Therefore, there can be no ambiguity concerning the matching of incomplete tokens. Furthermore, when an incomplete token arrives at an input node, it is matched against the stored full tokens of the corresponding memory. Therefore, the output token is more informative that the input one because it contains more variables.

Example 7.

Consider the rule DR2 of Example 3. In addition, assume that the tokens of Example 6 still reside at the two-input event memories. If the following message:

`delete_end(['Serres'])` ⇒ `41#arc`

is sent, then the negative token `[41#arc,'Serres']` is propagated to the right input of the intra-object event for class `arc`. There, it is matched with the stored tokens (only one match), deleted from the right memory and an incomplete negative token `[41#arc,Y,'Serres']` is propagated to the right-input of the inter-object event. In the right-input it is matched with the stored tokens, variable `Y` is instantiated to `'Kilkis'`, and the token is deleted from the right-input memory. Finally, an incomplete token `[X,'Serres']` is output.

The incomplete token is propagated to the deductive rule manager which "requests" the deletion of the partially instantiated derived object `path(start:X,end:'Serres')`. This partial instantiation is interpreted as *"delete all paths that end in node Serres"* which is not correct, because only paths that contain `41#arc` should be deleted.

[]

The last example shows that the only two-input event that is not entitled to deletion optimization is the last one of the network since the action or the conclusion of the rule might "project away" some of the OID (or other) variables. Consequently, the propagation of the "deletion" from the condition to the action is non-deterministic.

Practically, the execution or non-execution of the join is controlled by the presence of a rule object at the output of a two-input event. When the event outputs to a rule, then the join is performed. Otherwise the deletion optimization holds, and incomplete tokens are output.

5.4.2 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, and they are mapped into one or more chained intra-object events (two-input events) that correspond to the equivalent positive patterns. The first inter-object event, however, that is connected to the last node of the intra-object event that corresponds to the negative pattern behaves in a slightly different manner than a usual (positive) inter-object event. Such a two-input event that one of its input nodes maps a negative condition element is called a *negative event*.

Example 8.

If the `arc` and `path` objects of Example 3 are extended with the length of the corresponding distance, then the following deductive rule extends an existing `path` using only the shortest `arc`.

```
IF    P1@path(start='Thessaloniki',end:Y,length:L) and
      not P2@path(start='Athens',end:Y,length<L)
THEN  closest_big_city(small_city:Y,big_city:'Thessaloniki')
```

The negative condition pattern causes a normal intra-object event that corresponds to the positive pattern to be created. However, the inter-object event that connects the two intra-object patterns is not a normal, positive two-input event, but rather a negative one because its right input stems from a negative condition pattern (Figure 5).

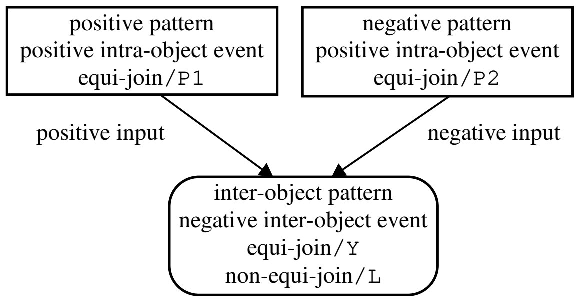

Figure 5. Example of a negative event.

Structurally, positive and negative events do not differ. 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 between positive and negative events is that the former have symmetrical inputs, i.e. the algorithms are the same regardless of the input source. Negative events, however, 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).

This is the reason why two negative intra-object events are never joined into an inter-object event. If this were done, no tokens would be propagated by the latter. Instead, the pre-compilation phase ensures that the order of intra-object patterns in the rule condition are such that a negative pattern is always joined with a positive one. More specifically, the complex event network is constructed in such a manner, that the "left" input of a negative event is always the "positive" input while the "right" input is the "negative" one.

When a negative event receives a positive token from its "positive" input and the input token does not match with any tokens stored at the "negative" memory, then a positive token is output. Otherwise nothing happens.

For instance, if the tokens `[39#path,'Katerini',470]`, `[11#path,'Theves',150]` are stored at the right memory of the negative event and the token `[48#path,'Katerini',70]` arrives at the positive input, it does not match with any of the previous tokens and the positive token `['Katerini']` is output. On the other hand, if the token `[96#path,'Theves',290]` arrives at the positive input, then it matches with the token `[11#path,'Theves',150]` and no output token is propagated.

On the other hand, when the “negative” input receives a positive token which matches some of the tokens stored at the “positive” memory, negative tokens are output. Of course, the negative tokens may not correspond to past positive tokens, but this check is left out for optimization reasons. When these "unverified" negative tokens are received by a successive event in the network, they are ignored. When no match exists for the input token, nothing happens.

For instance, if the tokens `[48#path,'Katerini',70]`, `[96#path,'Theves',290]` are stored at the left memory of the negative event and the positive token `[39#path,'Katerini',470]` arrives at the negative input then it does not match with any of the previous tokens and no output token is propagated. On the other hand, if the positive token `[11#path,'Theves',150]` arrives at the negative input, it matches with the token `[96#path,'Theves',290]` and the negative token `['Theves']` is output.

Finally, when a negative event receives a negative token from its “positive” input, the behavior is exactly the same as with the behavior of a positive event. However, when the negative token arrives at the “negative” input, the behavior is pretty much complicated. Specifically, if the input token matches some tokens of the “positive” memory, these are candidate output tokens. The latter are again matched against the “negative” memory. This is done in case of non-equi-joins where a single token can match more than one tokens of the opposite memory. The candidate output tokens that do <u>not</u> match any tokens of the “negative” memory are finally forwarded as positive output tokens.

For instance, if the token `[96#path,'Theves',290]` is stored at the left memory of the negative event, the tokens `[11#path,'Theves',150]`, `[19#path,'Theves',180]` are stored at the right memory and the negative token `[11#path,'Theves',150]` arrives at the negative input, it matches with the token `[96#path,'Theves',290]` of the left memory, and the latter is a candidate positive output token. However, `'Thessaloniki'` is still not the closest big city to `'Theves'` because there is one more path from `'Athens'` to `'Theves'` that is shorter than the distance between `'Thessaloniki'` and `'Theves'`. If the candidate token of the left memory is re-matched with the tokens of the “negative” memory, it is found that it cannot be propagated at the output because it matches with at least one token (`[19#path,'Theves',180]`) of the “negative” memory. Therefore, this candidate token is rejected, and since there is no other candidate, no tokens are output.

[]

5.4.3 The Compilation Process

The compilation process consists of 3 phases. During phase-1 (precompilation), the rule is parsed; the complex attribute references are transformed to simple attributes in multiple intra-object patterns (see Example 1); and the parse tree is optimized through re-ordering [4].

The main compilation phase consists of 2 stages. During stage-1, the condition is scanned in order to identify and create the primitive and logical events needed to monitor all the attribute patterns of the condition. This stage also collects information to assist stage-2 about the variables of the condition and action. During stage-2, the primitive and logical events are joined in pairs to form two-input events, based on the common variables. First the intra-object and then the inter-object events are constructed.

Finally, in phase-3, the ECA rule is created and linked with the last event of the discrimination network. At the end of rule creation, a rule activation phase recursively propagates activation messages from the rule down to the primitive events of the network which then query the database for objects that already exist and signal the tokens upwards to the network as if the objects were inserted now to an empty database. In this way, the network is correctly initialized with information about pre-existing data.

6. Extending/Optimizing E-DEVICE

E-DEVICE is very easily extendible due to the following reasons:

- the OO architecture allows the transparent modification of existing system components or the introduction of new ones;
- the rule managers of EXACT and DEVICE are quite general and flexible and allow the re-use and the partial modification of many of their components to support new rule types;
- the rule managers of E-DEVICE allow the modification of the rule semantics by changing the values of certain class attributes.

Deductive rules and rules for derived and aggregate attributes are some examples of extensions to the basic DEVICE system. New rule types are added in E-DEVICE by the following two steps:

Step 1. A new rule manager should be added as a subclass (direct or indirect) of the production rule manager (Figure 1). This gives the new rule manager inheritance of the default production rule semantics, such as the sharing of a common rule conflict set, the re-configurable conflict resolution criteria, and the rule action execution.

The conflict resolution criteria can be controlled by the corresponding class variables of the conflict set which can have `on` or `off` as their values. For example, the production and deductive rule managers have different conflict resolution criteria, as described in section 5.3.

When two rule types have a direct type-subtype relationship and exactly the same conflict resolution criteria, they can share a common conflict set. This sharing is controlled by an attribute of the corresponding rule manager. If the sharing attribute is set to `no`, the rule manager has an independent conflict set from its super-type(s) and is activated individually by the rule scheduler. When this variable is set to `yes`, the rule manager shares the conflict set of its first superclass that has its sharing attribute set to `no`. The existing rule managers of E-DEVICE are shown in Figure 6. The rule manager of persistent production rules shares the conflict set of the volatile production rules while deductive and non-materialized deductive rules maintain their own conflict set. Finally, derived and aggregate attribute rules have a common conflict set.

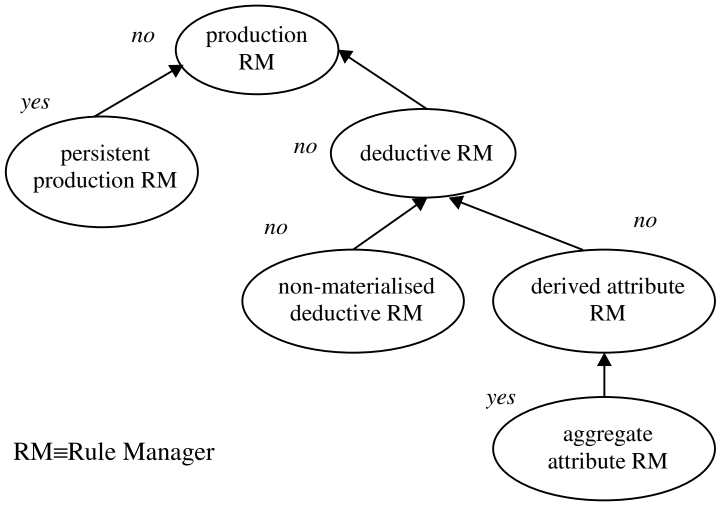

Figure 6. Sharing of conflict sets among the rule managers of E-DEVICE

Step 2. The new rule manager can be either an instance of the metaclass of the production rule manager or a new metaclass can be generated as a subclass of the metaclass of the production rule manager. The former solution gives the new rule type a rule creation semantics (parsing, pre-compilation, optimization, compilation, rule object generation etc.) similar to the one of production rules. The latter solution can be used when the new rule type partially inherits the rule generation semantics of production rules and partially re-defines them, as it is the case for deductive rules.

6.1 Optimization of the Complex Event Network

The event network requires a large storage space for the two-input memories and can duplicate the contents of the database in certain cases. In order to avoid such a waste of space, we introduced certain optimizations to the basic discrimination network that require less space at no or very little extra performance cost. These are the virtual and hybrid memories of two-input events.

It must be noted here that these optimizations are a compromise between complex events, performance, and space overhead. More specifically, the whole idea behind complex events is to have an incremental and autonomous mechanism that is able to detect whether multiple events have occurred in the database. However, for space optimization purposes and no extra performance cost, we are willing to sacrifice the autonomicity of complex event detection.

6.1.1 Virtual Memories

Virtual memories save storage space for rule conditions that include attribute patterns with no attribute tests. This optimization also increases the performance of matching since the storage management of large event memories requires some extra time that is not compensated by the smaller size of the joined relations because there is no selection associated with the node. Consider for example the following intra-object pattern:

```
A@arc(start='Thessaloniki',end:Y)
```

There is not a direct test associated with attribute `end`; therefore, nothing is gained by storing all the signaled `[A,Y]` tokens at the corresponding intra-object event memory. Instead, the values of the attribute `end` are directly retrieved from the original `arc` objects when required with no performance penalty.

The same principle applies to larger patterns of the condition, such as intra-object patterns:

```
P@path(start:X,end:Y) and
A@arc(start:Y,end:Z)
```

Both intra-object patterns consist of two attribute patterns with no direct attribute tests. There is no gain if all signaled tokens are stored at the memories of the successor inter-object event. Instead, the values of the attributes are directly retrieved from the database objects when required. In this example, none of the variables are stored. Therefore, there is 100% saving in storage space of two-input memories. Virtual memories can appear only at intra-object events or at the first successor inter-object event.

6.1.2 Hybrid Memories

Hybrid memories mix stored and virtual variable values in an input token of an inter-object event. Hybrid memories store only the absolutely necessary information and retrieve the rest of the token values from the original database objects. Consider the following rule condition:

```
A1@arc(start='Thessaloniki',end:X,length:L1) and
A2@arc(start:X,end:Y,road='Highway',length:L2) and
A3@arc(start:Y,end='Athens',length:L3)
```

This condition involves 3 related instances of class `arc` with various differing selection criteria. When each of the 3 intra-object events emanates an output token, it means that a certain `arc` has satisfied the corresponding intra-object pattern. This `arc` instance is uniquely identified by its OID; therefore, there is no need to store in the input memories of the subsequent inter-object events the rest of the variables that appear inside the pattern because these can be very easily retrieved later by accessing the original object.

The above concept is illustrated in Figure 7. This shows that tokens flowing from event to event are different from the tokens actually stored in the corresponding memories. In addition to the OID variables that are stored in every memory, the join variable of each inter-object event is also stored at the input memories, in order to ease the process of joining at the expense of a little extra storage space at each memory. The rest of the variables that flow into each inter-object event are virtual, i.e. they are not stored but directly accessed via messages from the corresponding objects. A non-OID variable can become both a stored and a virtual variable, depending on the inter-object event, whereas OID variables are always stored variables.

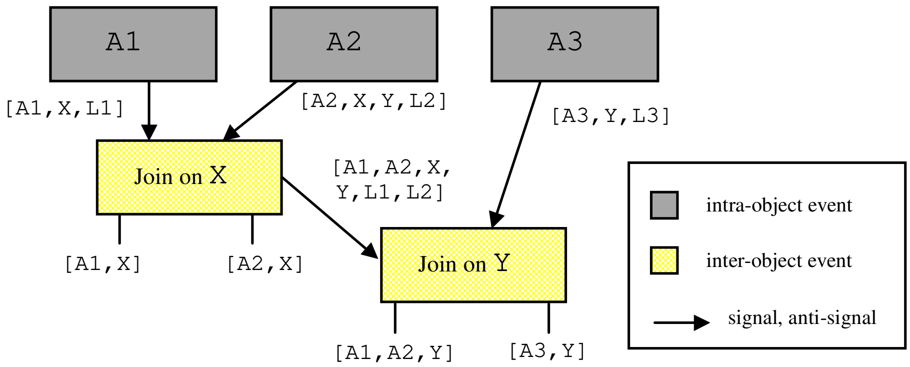

Figure 7. Hybrid memories of inter-object events

At run-time, first the stored variables are matched against the incoming token and upon success the virtual variables are retrieved and matched. Of course, this extra complication of token retrieval and matching has some extra overhead because tokens are matched in one step while the direct object access requires one message per attribute to be sent to the corresponding object, but the gain in storage requirements can be much more significant. Specifically, for the above condition, only 9 out of 16 variables are stored in the inter-object memory space.

6.2 Set-Oriented Rule Execution Scheme

The rule execution cycle we have considered so far is tuple-oriented which means that at each production cycle only one rule instantiation is considered. A possible optimization of this scheme is to gather together all the instantiations of the same rule and execute them in a single step. In this way, a significant reduction in the number of production cycles and the rule execution time can be achieved, as it is demonstrated in the performance section. However, it must be noted that the semantics of rule execution regarding conflict resolution is slightly changed due to the set-oriented nature of rule execution. Specifically, a case might arise where a rule that waits in the conflict set for a long time is promoted and executed due to a single update [23].

7. System Performance

In this section, we measure the performance of deductive rule execution in E-DEVICE, and we compare it to the one-to-many approach of [8]. For the tests, we use the definition of the `path` derived class in Example 9 which is similar to Example 3. However, it does not compute all the derivable `path` objects, but only those paths reachable from an initial node to all other nodes of the graph. The `arc` database we used comprises of the road connections between cities of Macedonia, Greece. The initial node is Thessaloniki, the capital of Macedonia. Three sub-graphs of the map of Macedonia have been used with increasing detail, i.e. more cities and road connections.

<u>Example 9.</u>

```
DR1:  IF    A@arc(start:thessaloniki,end:Y)
      THEN  path(start:thessaloniki,end:Y)
DR2:  IF    P@path(start:thessaloniki,end:Y) and
            A@arc(start:Y,end:Z\=thessaloniki)
      THEN  path(start:thessaloniki,end:Z)
```

[]

The ECA rules and events that are needed to maintain the same derived class under the one-to-many approach were manually generated and tested. The results are shown in Table 5 and Table 6. Columns marked *A*, *B* represent the materialized and non-materialized approaches of E-DEVICE, respectively. The keyword *set* indicates the set-oriented rule execution. Finally, columns marked with *C* represent the one-to-many approach. The numbers indicate user CPU time in seconds measured on a Sun Ultra Enterprise 3000 with Solaris 2.5.1. Notice that the tables show only derivation costs as the message sending and object creation costs of ADAM have been subtracted.

Table 5 shows the performance for bulk and single inserts. The set oriented approach is favored for bulk inserts since too many similar actions are executed in a single production cycle. More specifically, Table 7 shows the production cycles for each of the tests which explains why the set-oriented approach is significantly faster than the tuple-oriented one. The non-materialized approach is slower than the materialized one because the matching phase is separated from the object creation phase, and many pieces of code are redundantly re-executed. Furthermore, the one-to-many approach is slower than the E-DEVICE approach because of redundant condition checking during the matching phase (analyzed in section 4.1).

The single insert test shows that there is no significant difference between the tuple-oriented and the set oriented approaches since only one action at any cycle is executed. Furthermore, it is evident that the materialized approach is significantly faster than the non-materialized approach because of incremental matching. The materialized approach's only disadvantage is the storage space occupied by the discrimination network and the derived objects, whereas the non-materialized approach empties this space after query execution. Furthermore, the E-DEVICE (materialized) approach is faster than the one-to-many approach and scales-up more nicely. Redundant condition checking and joins computed between larger sets of instances in the one-to-many approach explains this, whereas the E-DEVICE approach pre-selects instances according to the tests present in the rule condition.

Table 6 shows the performance for deletions of a single `arc` that causes the deletion of 0 paths (alternative derivations) and paths of length 1, 2 and 3 arcs. The set-oriented approach is faster only at the deletion of the path of length 3 because there exist more than one anti-actions that can be executed in a single production cycle (Table 7). The one-to-many approach is significantly slower (up to 2 orders of magnitude!) because during the propagation of the deletion from the arcs to the paths, several paths that shouldn't really be deleted are deleted and must be re-derived [8]. This is shown in Table 7 where the production cycles for the one-to-many approach are shown in the

Cities/Arcs	Bulk Insert A	A (set)	B	B (set)	C	Single Insert A	A (set)	B	B (set)	C
7/18	6.90	5.00	10.03	8.44	8.99	0.92	0.92	12.02	10.05	1.39
13/38	18.21	10.16	25.10	18.32	18.82	0.95	0.92	28.69	20.96	1.50
28/88	64.59	24.87	96.03	57.78	47.90	0.97	0.98	103.08	62.82	1.79

Table 5. Performance comparison between the E-DEVICE and the one-to-many approach for inserts

	Delete path length 0			**length 1**	**length 2**	**length 3**
Cities/Arcs	*7/18*	*13/38*	*28/88*	*13/38*	*28/88*	*28/88*
A	0.59	0.59	0.57	0.88	1.87	1.90
A (set)	0.59	0.58	0.58	0.86	1.88	1.89
C	6.33	11.73	52.95	1.59	51.95	52.24

Table 6. Performance comparison between the E-DEVICE and the one-to-many approach for deletes

	Bulk Insert			**Delete path length 0**			**length 1**	**length 2**	**length 3**
Cities/Arcs	*7/18*	*13/38*	*28/88*	*7/18*	*13/38*	*28/88*	*13/38*	*28/88*	*28/88*
A	14	32	78	2	2	2	2	2	3
A (set)	4	5	6	2	2	2	2	2	2
C	1-1-3	1-1-4	1-1-5	4-2-2	4-2-4	12-2-5	2-1-1	13-2-4	14-2-4

Table 7. Production cycles for the various tests

following order: *delete-re-derive-insert*. The counter-based approach of E-DEVICE avoids the unnecessary deletion and re-derivation costs.

A noticeable advantage of E-DEVICE is that the deletion time (for deletion of 0 paths) does not scale-up with the size of the database because the search among the `path` objects is restrained among the ones that begin from Thessaloniki. The one-to-many approach, on the other hand, scales-up rapidly due to the increasing complexity of the road network and the associated deletions and re-derivations. The rest of the deletion tests clearly show that E-DEVICE is significantly faster at most of the cases, and the reasons are the same with the previous tests. As an exception, the deletion of the path of length 1 is "only" 50% slower because the deleted arc does not affect the connectivity of the graph.

8. Conclusions and Future Work

In this paper we presented the architecture of an extensible active Knowledge Base System, called E-DEVICE, that provides the infrastructure for the integration of multiple declarative rule systems into a single active OODB. E-DEVICE is a large extension of an older system, called DEVICE, that is based on the translation of production rules into one ECA rule which is triggered by a network of complex events. E-DEVICE allows the seamless integration of new declarative rule types by adding new rule managers and compilation schemes. This is achieved by generalizing the functionality of production rules. In the paper we described the integration of deductive rules using a materialized and a non-materialized approach, and rules for derived and aggregate attributes.

E-DEVICE has been fully implemented in the EXACT active OODB [11]. The performance of deductive rules using the E-DEVICE approach was measured to be faster than the one-to-many approach on incremental insertions and deletions while under a set-oriented rule execution scheme, it is also better for bulk inserts.

In the future, we plan to add more rule types such as integrity constraints. We will explore the possibility of translating a high-level functional constraint language, called CoLan [1], into production rules by providing constraint repair actions. Furthermore, the deferred “event-condition” coupling of DEVICE production rules provides the “right” framework for alleviating the strictness with temporary integrity violation that is inherent in the method-embedded integrity checking technique of [1].

Our current work includes the incorporation of the declarative rule facilities into a parallel OODB system, named PRACTIC [2], in order to provide a fast and flexible parallel active KBS. We are currently working on parallelizing an active rule system, examining issues such as rule execution concurrency, control and distributed conflict resolution.

Furthermore, we explore the integration of E-DEVICE as a main-memory add-on module to the object-oriented multidatabase InterBase* [26]. This integration will provide a system with the necessary functionality for data warehousing. Specifically, InterBase* will provide interoperability at the application level and high-level support for an integrated atomic commitment [27] while E-DEVICE will offer a declarative language for defining complex views over the source data and an efficient mechanism for materializing and self-maintaining those views [5].

9. Appendix A - Syntax of E-DEVICE Rules

```
<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> ::= [<variable>'@']<class>['('<attr-patterns>')']
<attr-patterns> ::= <attr-pattern>[','<attr-patterns>]
<attr-pattern> ::= <var-assignment> | <predicate>
<attr-pattern> ::= <attr-function>':'<variable> <rel-operator> <value>
```

```
<var-assignment> ::= <attr-function>':'<variable>
<predicate> ::= <attr-function> <predicates>
<predicates> ::= <rel-operator> <value> [{ & | ; } <predicates>]
<rel-operator> ::= = | > | >= | =< | < | \=
<value> ::= <constant> | <variable>
<attr-function> ::= [<attr-function>'.']<attribute>
<prolog_cond> ::= 'prolog' '{'<prolog_goal>'}'
<action> ::= <prolog_goal>
<derived_class_template> ::= <derived_class>‘(‘<templ-patterns>‘)’
<derived_attribute_template> ::= <variable>'@'{<class>}'('<templ-patterns>')'
<templ-patterns> ::= <templ-pattern> [‘,’ <templ-pattern>]
<templ-pattern> ::= <attribute>‘:’{<value> |
                                    <aggregate_function>'('<variable>')'}
<aggregate_function> ::= 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
<variable> ::= A valid Prolog variable
```

10. References

[1] N. Bassiliades and P.M.D. Gray, “CoLan: A functional constraint language and its implementation,” *Data & Knowledge Engineering*, Vol. 14, No. 3, 1995, pp. 203-249.

[2] N. Bassiliades and I. Vlahavas, “PRACTIC: A concurrent object data model for a parallel object-oriented database system,” *Information Sciences*, Vol. 86, No. 1-3, 1995, pp. 149-178.

[3] N. Bassiliades and I. Vlahavas, “DEVICE: Compiling production rules into event-driven rules using complex events,” *Information and Software Technology*, Vol. 39, No. 5, 1997, pp. 331-342.

[4] N. Bassiliades and I. Vlahavas, “Processing production rules in DEVICE, an active knowledge base system,” *to appear in Data & Knowledge Engineering*, 1997.

[5] N. Bassiliades, I. Vlahavas, A.K. Elmagarmid, and E.N. Houstis, *InterBaseKB: A Knowledge-based Multidatabase System for Data Warehousing*, tech. report, Dept. of Computer Science, Purdue University, W. Lafayette, Indiana, September, 1997.

[6] E. Bertino and B. Catania, “Static Analysis of Intensional Databases in U-Datalog,” *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Montreal, Canada, 1996, pp. 202-212.

[7] E. Bertino, B. Catania, G. Guerrini, and D. Montesi, “Transaction Optimization in Rule Databases,” *Proc. Int. Workshop in Data Engineering - Active Database Systems*, Houston, Texas, 1994, pp. 137-145.

[8] S. Ceri and J. Widom, “Deriving incremental production rules for deductive data,” *Information Systems*, Vol. 19, No. 6, 1994, pp. 467-490.

[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*, Vol. 39, No. 9, 1994, pp. 555-568.

[10] U. Dayal, A.P. Buchman, and D.R. McCarthy, “The HiPAC project,” *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 and A. Jaime, *EXACT: An extensible approach to active object-oriented databases*, tech. report, Dept. of Languages and Information Systems, University of the Basque Country, San Sebastian, Spain, 1994.

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

[13] C.L. Forgy, *OPS5 User Manual*, tech. report, Dept. of Computer Science, Carnegie-Mellon University, 1981.

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

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

[16] N.H. Gehani and H.V. Jagadish, “Active database facilities in ODE,” *Active Database Systems: Triggers and Rules for Advanced Database Processing,* J. Widom and S. Ceri, Eds., Morgan Kaufmann Publishers, 1996, pp. 177-206.

[17] G. Gottlob, G. Moerkotte, and V.S. Subrahmanian, “The PARK semantics for active rules,” *Proc. Int. Conf. On Extending Database Technology*, Springer-Verlag, Avignon, France, 1996, pp. 35-55.

[18] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, “Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals,” *Proc. IEEE Int. Conf. on Data Engineering*, 1996, pp. 152-159.

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

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

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

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

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

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

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

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

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

[28] I.S. Mumick, D. Quass, and B.S. Mumick, “Maintenance of data cubes and summary tables in a warehouse,” *Proc. ACM SIGMOD Int. Conf. on the Management of Data*, Tucson, Arizona, 1997, pp. 100-111.

[29] I.S. Mumick and O. Shmueli, “Universal Finiteness and Satisfiability,” *Proc. ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1994, pp. 190-200.

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

[31] N.W. Paton, “Supporting production rules using ECA rules in an object-oriented context,” *Information and Software Technology*, Vol. 37, No. 12, 1995, pp. 691-699.

[32] N.W. Paton, O. Diaz, and M.L. Barja, “Combining active rules and meta-classes for enhanced extensibility in object-oriented systems,” *Data & Knowledge Engineering*, Vol. 10, 1993, pp. 45-63.

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

[34] T. Sellis, C.-C. Lin, and L. Raschild, “Coupling production systems and database systems: A homogeneous approach,” *IEEE Trans. on Knowledge and Data Engineering*, Vol. 5, No. 2, 1993, pp. 240-255.

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

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

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

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

[39] J. Widom, “The Starburst Rule System,” *Active Database Systems: Triggers and Rules for Advanced Database Processing,* J. Widom and S. Ceri, Eds., Morgan Kaufmann Publishers, 1996, pp. 177-206.