A Deductive Object-Oriented Database System based on Active Rules*

N. Bassiliades** **and I. Vlahavas**

Dept. of Informatics, Aristotle University of Thessaloniki, 54006 Thessaloniki, Greece.
E-mail: {nbassili,vlahavas}@csd.auth.gr **URL:** http://www.csd.auth.gr/~plk

Abstract

This paper describes a Deductive Object-Oriented Database (DOOD) system that is built on top of an active Object-Oriented Database (OODB) system. The system, named DEVICE, uses the primitives of the latter, like active rules, simple and complex events, to integrate deductive and production rules. The integration is based on the emulation of deductive rules by special purpose *if-then-else* production rules that have been smoothly integrated into an active OODB. The DEVICE system supports thus multiple rule systems, like active (event-driven), production (data-driven) and deductive (goal-driven) rules into the same OODB system. The core of this multiple rule integration is: a) the mapping of each high-level rule into one event-driven rule, offering centralised rule selection control for correct run-time behaviour and conflict resolution, and b) the use of complex events to map the conditions of high-level rules and monitor the database to incrementally match those conditions. DEVICE is extensible because a) it re-uses the primitives of the host active OODB system to build the integration scheme, without introducing low-level data structures that do not blend well with the OO model and are not easily extensible, and b) the rule managers support general-purpose rule scheduling functions. In conclusion, DEVICE is a flexible Knowledge Base System (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.

1. Introduction

New applications of database technology, such as design and concurrent engineering applications, integration of heterogeneous, independent databases, applications involving multimedia data, etc., require advanced data representation and processing features, like deductive capabilities and object-orientation. Deductive Object-Oriented Databases (DOOD) constitute a fairly recent research area falling within the intersection of logic and databases. They reflect the growing need of both designers and users for a representation tool that is semantically richer than that provided by relational databases (RDB), as well as to the widely acknowledged benefits of using logic as a tool for the formalisation of both the static and the dynamic aspects of databases.

A deductive database [25] incorporates aspects of logic programming and thereby provides advantages over pure relational systems by bridging the gap towards knowledge bases. Deductive databases allow users, through the means of deductive rules, to deduce facts concerning data stored in the database. Deductive rules declaratively describe data in terms of other data, without an exact description of how new data are treated.

Object-Oriented Databases (OODBs) are an intersection of object-oriented (OO) ideas and conventional databases. They gained attention [3] mainly because they reduce the “semantic gap” be-

* This work is partially supported by EU under the INCO-COPERNICUS'96 project (ref. No. 960114).

** The first author is supported by a scholarship from the Greek Foundation of State Scholarships (F.S.S. - I.K.Y.).

tween real world concepts and data representation models. This one-to-one mapping helps the development of complex applications, like CAD, simulation, knowledge-based systems, etc.

Though these two formalisms have been regarded as in opposition [26], indeed they can be integrated within a single approach, which selects compatible features and merges them into a unique setting. The feasibility of such an approach has been shown both at a theoretical [2, 18] and at an implementation level [8].

This paper presents an overview of DEVICE, a DOOD that is based on the integration of deductive rules into an active OODB, by translating each deductive rule into special *if-then-else* production rules. Production rules are then emulated in the active OODB by means of complex events and ECA rules. In this way we achieve the smooth integration of low-level active rules with the high-level deductive rules by using the primitives of the active database and not introducing new low-level data structures. The integration of these two rule paradigms is also an active research area [16, 24, 29].

The structure of the paper is as follows: Section 2 refers to related work on DOODs and deductive rule integration in active database systems. Section 3 presents the deductive rule language, while section 4 describes the method for integrating deductive rules in DEVICE. Finally, section 5 concludes this paper with a summary of the main points and a discussion of future work.

2. Related Work

There are several approaches to the integration of deductive and object-oriented databases [12]. They mostly fall into three categories, a) object-oriented (OO) extensions to Datalog, like e.g. [1, 2], b) OO extensions to Prolog, like e.g. [28, 21, 19], and c) OO extensions to first-order logic programming [18]. The most notable disadvantage of approaches in categories a) and c) is that they represent theoretical departures from logic, therefore their usability and practicality in terms of object-oriented features is limited. On the other hand, approaches of extending Prolog with OO features usually bring full-featured OO systems, with the additional benefit of the existence of an underlying high-level language. However, Prolog is mostly used as a medium for meta-programming OO features and is not smoothly integrated with the OO model.

The DEVICE system follows a different approach by extending an OODB with deductive capabilities. This brings full OO functionality into the system and has the additional merits of using deductive rules for deriving new objects from the stored ones. The integration has been achieved by using the primitives of an active OODB, such as events and active rules in order to a) detect and record changes made to the database, and b) identify deductive rules that must be executed in response to these updates.

The integration of active and deductive databases is an active research area. Some systems, like POSTGRES [23] and Chimera [9] support only backward chaining Prolog-like deductive rules through data retrieval events. Most of the bottom-up evaluation approaches [16, 24, 29] try to incorporate events into a deductive database. However, these approaches depart from relational-deductive databases, where the notion of pure events is absent. In OODBs instead, event-driven rules are the most "natural" choice for generic rule support, since events conform to the message-passing paradigm of OO computation and every recognisable message/method can be a potential event. Therefore, event-driven rule execution can be very easily implemented as a "detour" from normal method execution.

In this paper we consider the case of incorporating deductive (and production) rules into a system that supports only event-driven rules by translating each high-level rule into one event-driven rule. The translation is based on the emulation of a deductive rule through a special *if-then-else* production rule. This method is also used in the RDL1 system [17], under a more user-controlled and low-level deductive rule scheme. Production rules in DEVICE are integrated into the active OODB using the same scheme. The conditions of both high-level rule types are incrementally matched using a complex event network that resembles the RETE-like discrimination networks met in production systems.

Similar attempts to integrate deductive and production rules into an active OODB system with event-driven rules can be found in [10, 22] respectively. They follow a simpler approach, by emulating each high-level rule using many event-driven rules. The main advantages of our approach are:

a) easier rule maintenance, b) centralised rule selection and execution control, c) straightforward implementation of traditional conflict resolution strategies of KBSs, and d) net effect of events. Furthermore, the performance comparison of our approach with [22] (in [7]) showed that DEVICE is faster for incremental insertion and deletion of data, while it can also be considerably faster for bulk data loading under a set-oriented rule execution scheme.

DEVICE has been designed for implementation on any suitably extensible active OODB system, to support the desired functionality. It has been implemented on EXACT [11], a descendant of ADAM [21]. ADAM is an OODB system built on top of Prolog, so the data definition, query and manipulation language is Prolog extended with message-passing predicates.

3. Deductive Rule Language

This section presents the deductive rule language of DEVICE. Deductive rules are composed of *condition* and *conclusion*, whereas the *condition* defines a pattern of objects to be detected over the database and *conclusion* is a derived class template that defines the objects that should be in the database when the condition-part is true.

Example 1. The following deductive rule:

```
DR1:   IF A@arc(start=1,end:Y) THEN path_from_one(end:Y)
```

defines that an object with attribute `end=Y` exists in class `path_from_one` if there is an object `A` in class `arc` with attributes `start=1`, `end=Y`.

Class `path_from_one` 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 derived objects is a union of the objects derived by the two rules. For example, the transitive closure of the set of nodes reachable from node `1` is completed with the following (recursive) rule:

```
DR2:   IF     P@path_from_one(end:Y) and A@arc(start:Y,end:Z\=1)
       THEN   path_from_one(end:Z)
```

The condition of a rule is an *inter-object* pattern, that consists of the conjunction of one or more (either positive or negative) *intra-object* patterns. Disjunction of intra-object patterns is not included in the initial design for simplicity of implementation, since it is implied by the presence of multiple rules.

The intra-object patterns consist of one or more *attribute* patterns. The only intra-object pattern of rule `DR1` denotes an instance `A` of class `arc` with attribute `start` equal to `1` and attribute `end` equal to `Y`. The first of the intra-object patterns of rule `DR2` denotes an instance `P` of class `path_from_one` with attribute `end` equal to `Y`, while the second intra-object pattern describes an instance of class `arc` with attribute `start` equal to the attribute `end` of the previous `path_from_one` object. Notice that we require the attribute `end` of the `arc` object not to cycle back to node `1`.

Variables in front of 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 conclusion of the rule. The values delivered by variables can be both object references and normal values, e.g. integers, strings. The variables are expressed as valid Prolog variables.

There can also be negated intra-object patterns in the condition. A negated intra-object pattern denotes a negative condition that is satisfied when no objects in the database satisfy the corresponding positive intra-object pattern.

Example 2. The following rule does not allow a path (of an arbitrary length) to be created if a path of length 2 exists:

```
IF     P@path_from_one(end:Y) and A@arc(start:Y,end:Z\=1)
       and not A1@arc(start=1,end:Y)
THEN   path_from_one(end:Z)
```

We notice that only safe rules [41] 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 (e.g. variable `A1`).

Notice the logic-like condition language that is used to declaratively specify the state of the database. The actions of production rules are expressed in the procedural ADAM [21] language. The conclusion of the rule is described by the derived class template which consists of attribute-value pairs, where the value can either be a variable that appears in the condition or a constant.

4. Deductive Rule Integration in DEVICE

Deductive and production rules are compiled to Event-Condition-Action (ECA) rules, in order to be constantly monitored by the active database. More specifically the deductive rule is compiled into a special *if-then-else* production rule that in turn is compiled into an ECA rule. The condition of the production rule is compiled into a complex event network, which is associated with the event-part of the ECA rule, while the then-part of the production rule is the action-part of the ECA rule. The compilation task is not trivial and requires sophisticated algorithms [5]. This section overviews the deductive and production rule integration aspects of DEVICE.

4.1. Compilation of Deductive Rules into Production Rules

The integration of deductive rules in DEVICE is achieved by mapping the deductive rule semantics on production rules. More specifically, deductive rules are in the following form:

```
IF condition THEN conclusion
```

where `conclusion` (also called the *head* of the rule) is a pattern of a derived object. In DEVICE we followed the materialised approach [10, 14], where each derived object is stored in the database for future use. Therefore, when the condition of the deductive rule is satisfied then the object described in the head of the rule 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. According to this procedural interpretation, deductive rules can be emulated using production rules, as in RDL-1 [17].

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 [26].

Furthermore, when the condition of a deductive rule is false then the derived object of the head must be removed from the database, otherwise the database will be inconsistent. Before this is done, however, it must be ensured that the derived object is not deducible by another rule instantiation. For this reason, we use a counter mechanism which stores the number of derivations of an object [14]. If the derived object has a counter equal to 1, then it is deleted, otherwise the counter is only decreased by 1.

The creation of a derived object is implemented by the `action` of the production rule, which is derived at compile time from the `conclusion`:

```
IF condition THEN derive(object)
```

where the method `derive` is based on the existence or not of `object`, as described earlier.

The deletion of a derived object can be modelled by another production rule as follows:

```
IF object and not(condition) THEN un-derive(object)
```

where the method `un-derive` deletes the `object`, subject to the check of the counter as described earlier.

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. The deletion of a derived object cannot be modelled by usual production rules though, therefore we extended the production rule with the `anti_action` (ELSE part) that hosts the derived object deletion algorithm. Using this extended production rule scheme a deductive rule is translated into:

```
IF condition THEN derive(object) ELSE un-derive(object)
```

Derived Classes

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` attribute we described above.

Derived classes are instances of the `derived_class` meta-class, which defines their common structure and behaviour, like the method `get_derived` that retrieves derived objects according to their attribute values. Derived classes allow only retrieval operations on their instances, since explicit creation, deletion or modification of derived objects might violate their integrity.

4.2. Compilation of Production Rules into ECA Rules

The main idea behind production rule compilation is that production rules are in the form:

```
IF condition THEN action
```

while ECA rules are in the form:

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

According to [27] there exist ways to translate the former to the latter or, in other words, to embed production rules into an active database system using the primitives of the latter.

The translation strategy of [10, 22] uses simple (primitive) events in the event specification of an active rule and one production rule is translated into many ECA rules. Each ECA rule is triggered by a different event, derived from a single condition element of the condition-part of the production rule. The condition-part of the ECA rule is the same as the condition-part of the production rule, minus the event. For example consider the following (abstract) production rule:

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

We assume that the primitive events e_a, e_b, e_c detect the insertion of the corresponding data items. The production rule `P1` is translated into the following 3 ECA rules:

```
A1: ON ea IF b & c THEN <action>
A2: ON eb IF a & c THEN <action>
A3: ON ec IF a & b THEN <action>
```

Therefore, from 1 production rule 3 ECA rules are derived to monitor the database for the pattern of the production rule condition. There are certain problems associated with this one-to-many approach, identified in the following paragraphs.

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.

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 fired, since their conditions are not satisfied. Finally `A1` is triggered, its condition is satisfied and the rule fires. This behaviour 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.

Redundant action execution. Now we 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 fired, 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, which is not the correct behaviour.

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.

The DEVICE translation method uses complex events instead, and translates the production rule `P1` to only 1 ECA rule:

```
CA1: ON ea & eb & ec [IF true] THEN <action>
```

The above ECA rule is equivalent to the production rule P1, provided that the desired operational semantics are supported by the complex event manager of the OODB. The incremental rule matching is achieved through the incremental detection of the equivalent complex event. The condition part of the ECA rule is always `true`, because all conditions tests have been incorporated into the complex event. Our approach avoids all the problems inherent to the one-to-many approach.

4.3. Rule Condition Matching

The efficient matching of production rules is usually achieved through a discrimination network [13, 20]. DEVICE smoothly integrates a discrimination network into an active OODB system, using its primitives. The discrimination network of DEVICE is formed by complex events, where each node of the network is mapped onto a complex event object of the active database system (fig. 1). In this way, the discrimination network is uniformly integrated into the OODB system, as a set of first class objects. More details and examples of both the compilation and matching processes can be found in [5, 6].

Primitive Events

The primitive database events detected by the active database system are input sources of the DEVICE network. 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. The primitive events are detected when the corresponding method is evoked, thus when a message is sent to an object. Actually, only the events raised *after* the execution of the corresponding methods are monitored, since only successful data modifications are interesting.

When a primitive event is *signalled* it means that a certain database state has been reached. These semantics is usually associated with the `put_` type events, (e.g. `put_start`) which correspond to messages that *insert* data in the database.

On the other hand, `delete_` type events correspond to methods that delete data from the database (e.g. `delete_start`). The occurrence of such events denotes that a certain pattern in the condition of the production rule that was previously present in the database is no longer valid. To model such semantics we introduce *anti-signalling*. We notice that `update_` type events are emulated by signalling a `delete_` type event followed by a `put_` type event.

When the event manager receives a signal (anti-signal) it forwards the positive (negative) token with the message parameters to the successor network nodes (subscribed objects) via the corresponding `checking` (`anti_checking`) method. These methods process internally the input tokens and if appropriate they construct output tokens. The event manager collects those tokens and repeats the loop. Primitive events output a 2-place token: the first value holds the object identifier (OID) of the message recipient object, while the second holds the parameter(s) of the message.

Logical Events

Logical events perform simple attribute tests and they are only raised when the associated condition is satisfied. In DEVICE, logical events map attribute pattern tests and they are signalled by primitive events to perform a check on the second value of the input token. If the check is successful

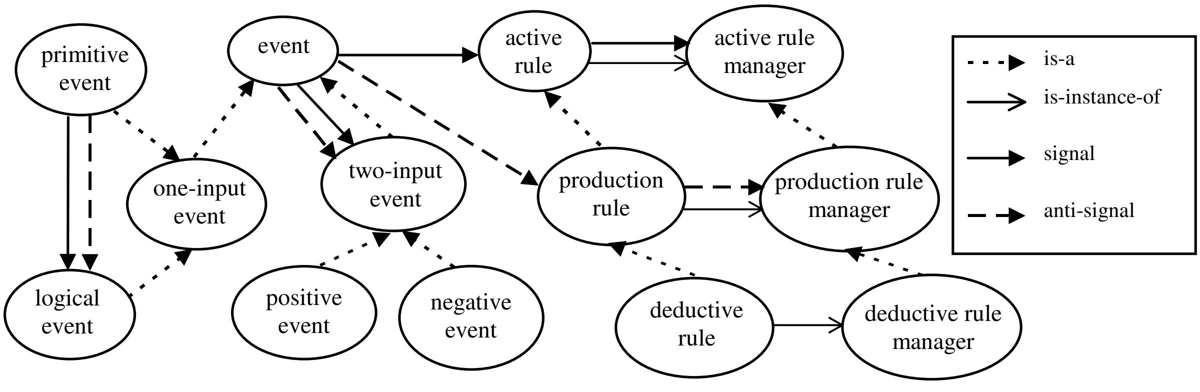

Figure 1. Event and rule objects of DEVICE

an output token is propagated to the event manager.

Two-input Events

The primitive and logical events are joined in pairs, into a two-input event. Furthermore, two-input events are also joined in pairs, into another two-input event, and so on so forth. We distinguish between 2 types of two-input events based on the relationship between the joined inputs. The two-input events that contribute to the detection of an intra-object pattern are called intra-object events and the common variable of the join is the OID of the message recipient objects.

Multiple intra-object patterns form an inter-object pattern and the corresponding two-input events are called inter-object events. The common variables of the join are derived from the shared variables in the rule condition. We notice that joins on any attribute are allowed, and not only on object references, as in [22].

If the rule condition contains more than two intra-object patterns, then more inter-object events are required. 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 the left and the right inputs. A memory associated with each input stores the incoming tokens with a counter that indicates how many times the token has already been received. When a positive token arrives through the `checking` message, it is searched at the associated memory. If it does not exist, then it is added with its counter equal to 1 and it is joined with the tokens of the opposite memory. If it already exists then its counter is increased by 1, but no join is performed. The join produces one or more positive output tokens according to a pre-compiled pattern.

Tokens describe states of the database therefore they must persist beyond the end of the transaction. Tokens are only explicitly deleted reflecting deletions in the database and they are not consumed when joined. The deletion of tokens is triggered by the detection of the primitive `delete_` type events and the propagation of the anti-signals throughout the network.

When a two-input event receives a negative token, through the `anti_checking` message from the event manager, it searches the appropriate memory for the token. If its counter is more than 1, then it is decreased by 1 and the anti-signal is not forwarded further. If the counter equals to 1, then the token is deleted from the memory and a negative output token is constructed. The construction of the output token depends on the type of rule the condition participates in. If the condition "belongs" solely to a production rule then no join is performed and the output token is incomplete. This is not valid for deductive rules because it can cause problems, since the object to be deleted is partially matched and the deletion process is non-deterministic.

To correct this behaviour we extended the structure of two-input events by an attribute that controls the delete-token optimisation. When a two-input event participates in the condition of a deductive rule, this attribute is set to `off` and the *anti-checking* method <u>will</u> perform joins at deletions. In this way two-input events are used for both production and deductive rules.

Negative Condition Elements

Negative intra-object patterns denote conditions that should <u>not</u> 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. The last intra-object event (of the negative pattern) is connected to an inter-object event, which behaves differently 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*.

Structurally, positive and negative events do not differ. However, their behaviour is different because the detection of the intra-object event at the “negative” input indicates that the (negative) inter-object pattern does not occur and vice-versa. The “negative” input does not contribute values to the output token of the inter-object event, because the negation is not constructive and it stands only for value testing (safety requirement).

When a negative event receives a *checking* message from its “positive” input and the input token does <u>not</u> match with any tokens stored at the “negative” memory, then a positive output token is constructed. If at least one match exists then nothing happens. On the other side, when the “negative” input receives a positive token that matches some of the tokens stored at the "positive" memory, negative output tokens are constructed from the stored ones, i.e. the input is inverted.

Furthermore, when a negative event receives a negative token from its "positive" input, the behaviour is exactly the same with that of a positive event. However, when the negative token comes from the "negative" input the behaviour is pretty much complicated. Specifically, if the input token matches some tokens of the "positive" memory, then these are candidates to be forwarded as positive output tokens. However, the candidate tokens must again be matched against the "negative" memory. This is needed for the case of non-equi-joins, where a single token can match more than one tokens of the opposite memory. The candidate output tokens than do not match any tokens of the "negative" memory are finally signalled as output tokens.

4.4. Deductive Rule Execution Control

When the event signalling loop does not detect any more complex event occurrences, the set of all event occurrences is forwarded to the deductive rule manager (fig. 1). The latter filters those event occurrences that activate deductive rules and forwards them to the generic rule manager to be scheduled for firing.

The rule manager maintains a set-valued attribute, traditionally called the "conflict set", that stores rule instantiations. When the rule manager receives a positive token it adds it to the conflict set, while a negative token causes the deletion of the corresponding positive token from the conflict set, if it exists, otherwise the negative token is also inserted in the conflict set.

When multiple rule instantiations are placed in the conflict set there is an ambiguity concerning the number and the order of rules to be executed. In DEVICE we incorporated three deductive rule execution control strategies into the priority mechanism of the EXACT system [11].

More specifically, the first control strategy removes from the conflict set the instantiation tokens of the rules that have been fired, in order to avoid infinite loops. This strategy is controlled from the class variable `refractoriness` [6] of the deductive rule manager that is set to `off`.

The second control strategy models the set-oriented semi-naive evaluation of deductive rules [25] by navigating the rule search space in a breadth-first or iterated manner. This strategy is controlled by the class variable `recency` of the deductive rule manager that is set to `off`. This causes the newly derived rule instantiations to be appended to the conflict set, which is a normal Prolog list, and to be considered only after the old ones [6].

The last control strategy implements the *stratification* of rules with negation [25]. When a deductive rule is created the derived classes that appear in the condition are collected along with their stratum number (i.e. their order of evaluation). The algorithm of [25] checks whether this 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 by the priority mechanism of EXACT to resolve rule selection at run-time.

This number is used by the system on rule creation to place the rule (NOT the rule instantiation) into a partially ordered set of all rule OIDs, called `ordered_rule_list`. This set is used at run-time to choose rule instantiations. The ordered rule set is intersected with the conflict set when the DEVICE system is ready to select and execute one rule from the conflict set. Therefore, this strategy alters at selection time the rule instantiation order of the conflict set according to the rule order of the `ordered_rule_list` attribute. The final totally ordered rule instantiation set is not known until the very last moment. The first rule instantiation in the above set is selected for firing.

After the rule manager selects a deductive rule for firing, it trivially checks the `true` condition and the action part of the corresponding production rule is scheduled for execution. The action is executed as a compound Prolog goal in "immediate" Condition-Action coupling mode.

On the other hand, the "deferred" EC coupling mode [11] is used in order to be able to execute multiple actions in one rule, without prematurely triggering and executing other rules in the medium. Events generated due to action execution are detected and forwarded through the network but no rule execution is performed. At the end of the transaction the rule manager starts the rule selection and execution sequences we described earlier.

In DEVICE the rule evaluation cycle is initiated either at the end of the transaction or at intermediate user-specified *checkpoints*. After the first rule is selected and executed the rule manager continues to the next evaluation cycle by considering all the previous rule instantiations plus any new ones that have been produced by the execution of the rule actions. This cycle continues until a *fix-point* is reached, where there are no more rule instantiations left in the conflict set.

5. Conclusions and Future Work

In this paper we presented DEVICE, a deductive and active OODB system that integrates deductive and OODB systems following a novel research line. More specifically, deductive rules are integrated into an active OODB as active rules that maintain the consistency of materialised OODB views, also called *derived classes*. The integration is based on the emulation of deductive rules by special *if-then-else* production rules. Production rules have been integrated into the EXACT active OODB [11] by using the primitives of the system [5, 6], namely simple events and ECA rules. The activation of production rules is incrementally detected by a network of complex events that plays the role of the discrimination network of production systems [13].

DEVICE is an extensible active Knowledge Base System that supports multiple rule systems into a single active OODB [7]. It has been fully implemented in the EXACT active OODB [11], which is an extension of ADAM [21], an OODB in Prolog. The fact that the core OODB system is a Prolog system extended with OO features does not have anything to do with the deductive features of DEVICE, which are implemented using the events and active rules of the system. Therefore, DEVICE could be implemented on top of any suitably extensible active OODB system.

The performance of deductive rules using the DEVICE approach was measured to be faster than the one-to-many approaches [10, 22] on incremental insertions and deletions, while it can also be better on bulk data loading for selective rule conditions. The performance can improve further by extending the idea of virtual nodes. Concerning the efficiency of the DEVICE network, we plan to support alternative optimised discrimination networks [15, 20], in addition to the RETE-like network we now support. Since all the above are just variations of the basic RETE network, it is fairly easy to implement them by extending the compilation scheme and the run-time behaviour of the complex event nodes.

We are currently pursuing the integration of these high-level rule facilities into the PRACTIC parallel OODB system [4] to provide a fast and flexible parallel active KBS. However, before this is done certain aspects of distributing and/or parallelizing an active system, like concurrency, must be resolved.

References

[1] S. Abiteboul and S. Grumbach, COL: A logic-based language for complex objects, in J.W. Schmidt, S. Ceri and M. Missikoff, eds., Advances in Database Technology - EDBT'88, Int. Conf. on Extending Database Technology, LNCS 303, Springer-Verlag, pp. 271-293, 1988.

[2] S. Abiteboul and P.C. Kanellakis, Object identity as a query language primitive, *Proc. Int. SIGMOD Conf. on the Management of Data*, 1989.

[3] M.P. Atkinson et al., Object-oriented database system manifesto, *Proc. 1st Internat. Workshop on Deductive and Object-Oriented Databases* (North-Holland, 1990).

[4] N. Bassiliades and I. Vlahavas, PRACTIC: A concurrent object data model for a parallel object-oriented database system, *Information Sciences* 86(1-3) (1995) 149-178.

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

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

[7] N. Bassiliades and I. Vlahavas, E-DEVICE: An extensible active knowledge base system with multiple rule support, *submitted for publication* (1997).

[8] F. Cacace et al., Integrating object-oriented data modelling with a rule-based programming paradigm, *Proc. Int. SIGMOD Conf. on Management of Data*, 1990.

[9] S. Ceri, P. Fraternalli, S. Paraboschi and L. Tanca, Active rule management in Chimera, in *Active Database Systems: Triggers and Rules for Advanced Database Processing*, eds, J. Widom and S. Ceri, Morgan Kaufmann (1996) 151-176.

[10] S. Ceri, and J. Widom, Deriving incremental production rules for deductive data, *Information Systems* 19(6) 467-490 (1994).
[11] O. Diaz and A. Jaime, EXACT: An extensible approach to active object-oriented databases, Technical Report, University of the Basque Country, San Sebastian, Spain (1994).
[12] A.A. Fernandes, N.W. Paton, M.H. Williams and A. Bowles, Approaches to deductive object-oriented databases, *Information and Software Technology*, 34(12) (1992) 787-803.
[13] C.L. Forgy, Rete: A fast algorithm for the many pattern/many object pattern match problem, *Artificial Intelligence* 19 (1982) 17-37.
[14] A. Gupta, I.S. Mumick and V.S. Subrahmanian, Maintaining views incrementally, *Proc. ACM SIGMOD Int. Conf. Management of Data* (1993) 157-166.
[15] E.N. Hanson, The Design and implementation of the Ariel active database rule system. *IEEE Trans. Knowledge & Data Engineering* 8(1) (1996) 157-172.
[16] J.V. Harrison and S.W. Dietrich, Integrating active and deductive rules, *Proc. 1st Int. Workshop on Rules in Database Systems* Edinburgh, Scotland, Springer-Verlag (1994) 288-305.
[17] J. Kiernan, C. de Maindreville and E. Simon, Making deductive databases a practical technology: A step forward, *Proc. ACM SIGMOD Int. Conf. on the Management of Data,* (1990) 237-246.
[18] M. Kifer and J. Wu, A logic for object-oriented programming (Maier's O-Logic revisited), *Proc. Int. Conf on Principles of Database Systems*, 1989.
[19] F.G. McGabe, *Logic and Objects*, Prentice Hall, 1992.
[20] D.P. Miranker, TREAT: A better match algorithm for AI production systems, *Proc. AAAI*, Seattle, WA, Morgan Kaufmann (1987) 42-47.
[21] N.W. Paton, ADAM: An object-oriented database system implemented in Prolog, *Proc. 7th British National Conf. on Databases* CUP (1989) 147-161.
[22] N.W. Paton, Supporting production rules using ECA rules in an object-oriented context, *Information and Software Technology* 37(12) (1995) 691-699.
[23] S. Potamianos and M. Stonebraker, The POSTGRES rule system, in *Active Database Systems: Triggers and Rules for Advanced Database Processing*, J. Widowm and S. Ceri (Eds.), Morgan Kaufmann Publishers (1996) 43-61.
[24] E. Simon, J. Kiernan and C. de Maindreville, Implementing high level active rules on top of a relational DBMS, *Proc. 18th Int. Conf. on Very Large Databases,* Vancouver, Canada, Morgan Kaufmann (1992) 315-326.
[25] J. Ullman, *Principles of Database and Knowledge-Base Systems*, Vols. I-II, Rockville, Maryland: Computer Science Press (1989).
[26] J. Ullman, A comparison between deductive and object-oriented database systems, in *Deductive and Object-Oriented Databases, Second Int. Conf. DOOD'91, Munich*, C. Delobel, M. Kifer and Y. Masunaga (Eds.), Springer Verlag, Berlin (1991) 263-277.
[27] J. Widom, Deductive and active databases: Two paradigms or ends of a spectrum?, *Proc. 1st Int. Workshop on Rules in Database Systems,* Edinburgh, Scotland, Springer-Verlag (1994) 306-315.
[28] C. Zaniolo, Object identity and inheritance in deductive databases - An evolutionary approach, in W. Kim, J.-M. Nicolas and S. Nishio, eds., *Deductive and Object-Oriented Databases (1st Int. Conf. DOOD'89, Kyoto)*, Elsevie, Science Press, 1990.
[29] C. Zaniolo, A unified semantics for active and deductive databases, *Proc. 1st Int. Workshop on Rules in Database Systems,* Edinburgh, Scotland, Springer-Verlag (1994) 271-287.