

Supporting Production Rules Using ECA-Rules In An Object-Oriented Context

Norman W. Paton

Department of Computer Science,
University of Manchester,
Oxford Road, Manchester M13 9PL
e-mail: norm@cs.man.ac.uk

Abstract. This paper presents an approach to implementing production rules for object-oriented databases (OODBs). The approach builds upon earlier work on production rule algorithms for relational databases, and exploits fundamental differences in the structuring mechanisms employed by OODBs. An implementation is described whereby the production rules are mapped onto event-condition action rules for execution. It is shown how the resulting implementation has minimal space overheads, and a time performance close to that of the widely used TREAT algorithm which uses significantly more space.

Keywords: Production rules, object-oriented databases, active databases.

1 Introduction

The area of active databases has emerged in recent years as a mainstream research direction for advanced database systems. Active functionality, normally expressed using Event-Condition-Action rules (ECA-rules), can be used to support time constrained or real time applications (commodity trading, traffic management, plant monitoring), or to implement extensions to the database system which support integrity constraints [9, 15], views [10], behaviour sharing [16], distribution [11], or advanced modelling constructs [30]. An ECA rule lies dormant until it is triggered by the occurrence of an *event*, which may be internal (e.g. a message send, an update operation) or external (e.g. from a clock or a temperature gauge) to the database. When an event has occurred that is of relevance to the rule, the *condition* of the rule is evaluated to examine the context within which the event has taken place, and if true then the *action* of the rule is executed. Different aspects of support for ECA-rules are presented in [32].

Active database systems, however, have much in common with production systems (which support condition-action rules), and some active databases provide direct support for production rules [27, 25]. For certain tasks which monitor the state of the database, such as integrity constraint checking or maintenance of derived data, many ECA-rules may be required to express the same functionality as a single production rule. Thus systems which support

only ECA-rules may provide cumbersome solutions to certain tasks which have been felt to be important applications of active functionality. While many proposals have been made for active OODBs [13, 22, 17, 12, 29], none of these support production rules. There thus seems to be a need for work on the integration of production rule facilities with OODBs. This paper addresses two issues of relevance to this task – the way in which such functionality can be implemented, and the performance which can be expected from the resulting system.

A major focus for research on the integration of production rule systems with relational databases has been upon efficient implementation, as approaches which were effective in main-memory production systems (such as Rete [20]) often do not scale well for use with large databases. This paper presents an implementation strategy for OODBs which compiles production rules into ECA rules, and shows that the performance of this approach is competitive with the performance of recently proposed algorithms for supporting production rules in relational systems. Furthermore, this approach to implementation facilitates the straightforward extension of OODBs supporting ECA rules with production rules.

The paper is organised as follows: section 2 outlines related work on active databases and production rule algorithms; section 3 outlines the principal characteristics of production rule algorithms for relational databases, and indicates how different priorities may exist in OODBs; section 4 shows how production rules can be implemented in an OODB using ECA-rules, and outlines performance issues; section 5 presents a number of issues which are raised by the approach taken in the paper; section 6 presents some conclusions.

2 Related Work

2.1 Active Databases

Most recent active database systems are based upon Event-Condition-Action rules (ECA-rules), early work on which was carried out in the HiPAC project [13]. The separation of the event as a distinct component in the structure of a rule was carried out for a number of reasons, including efficiency of event testing, increased execution model flexibility, and the possibility of monitoring external events. Proposals have been made for ECA-rule systems which operate over both relational [35, 37] and object-oriented [13, 23, 17, 12, 29] databases, and recent work has focused upon event specification languages [24, 21], transaction models [6], and formal specification [8, 2].

A number of proposals have blurred the distinction between ECA-rules and production rules. For example, in Starburst [37] rules have a distinct event component, but the occurrence of an event does not cause any immediate change to the flow of control. In Ariel [25] the specification of the event is optional, and it is argued that this enables many applications of active functionality to be described more concisely than in pure ECA-rule systems.

The work presented in this paper is complementary to the research mentioned above, in that it shows how a production rule system can be implemented using an ECA rule system, thereby providing an efficient implementation of production rules in an object-oriented context.

2.2 Production Rule Algorithms

Production systems can be used to implement certain aspects of database behaviour using condition-action rules of the form:

```
if <condition> do <action>
```

Such rules are then processed in the context of a recognise-act cycle:

```
match
while (conflict set not empty) do
  conflict resolution
  act
  match
end-while
```

In such a cycle, the **match** phase identifies rules with conditions that are true with respect to the state of the database, and adds them to the **conflict set**, which is essentially a queue of activated rules waiting to be fired. The **conflict resolution** step selects a single rule from the conflict set for further processing in the **act** phase, which executes the statements in the action of the selected rule.

A naive evaluation of the **match** phase would evaluate the condition of every rule to find which rules should be considered for processing. Such an approach would be prohibitively expensive, and is not necessary, as only a small part of the database over which each condition acts is likely to change during each cycle. Various techniques have been proposed which remove the need to re-evaluate rule conditions, including [20, 28, 34].

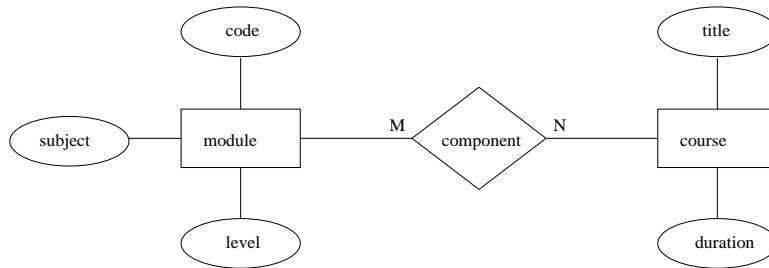


Figure 1: Entity/Relationship for course database

In what follows, the schema depicted by the E/R diagram in figure 1 will be used in examples to illustrate the various approaches. In the relational model, the relationship between a *module* and the *courses* of which it is part can be represented by the relations `module(code,subject,level)`, `course(title,duration)` and `component(title,code)`. A typical production rule (in Ariel notation [25]) relating to these relations could monitor level 4 modules which are part of **Computing** courses:

```
if crs.title = "Computing" and crs.title = cont.title and
   mod.level = 4 and mod.code = cont.code
  from crs in course, cont in component, mod in module
do <action>
```

The query which is used to express the condition involves a join of the `component` relation with both `course` and `module`.

The principal notion behind the Rete [20] matching algorithm which was originally proposed for implementing main-memory OPS-5 production rule systems is that, by storing the partially computed condition of a rule between cycles of rule execution, the effect of *changes* to the database on the conflict set can be computed with minimal additional effort. The partially computed condition of a rule is often stored in a graph structure known as a *discrimination network*. The example rule above can be represented by the Rete discrimination network in figure 2.

The Rete network has a number of different node types – a *root* node serves as the entry point to the network; below the root node are a number of nodes which represent the base tables; below any node which represents a stored table, there can be zero or more filter nodes each of which applies a single condition to the relation; after a filter has been applied, the tuples (or references to the tuples) which have satisfied the condition in the filter are stored in α memory nodes; a node with two parents performs a join on its parent relations, the result of which is stored in a β memory node; at the bottom of the tree is the result node which holds the conflict set.

The Rete match phase operates by passing information down through the network from the root. For example, if a new *course* tuple is entered into the database, it is tested to see if it has a *title* equal to *Computing* – if so, it is stored in the α memory node *alpha1*. The tuple is then joined with the α memory node *alpha2*, and if tuples result from the join, they are stored in the β memory node *beta1*. Any such new tuples are then joined to the α memory node *alpha3* to yield additional data for the conflict set.

While Rete has been used in commercial production-rule systems, there are a number of problems, especially when large databases are used: the space overhead is high, with both α and β memories storing the intermediate results of computation, and maintenance of such results, particularly on deletion of data, imposes a significant overhead. Recent studies have shown that, Rete is often not the algorithm of choice for large production rule systems [28, 7, 36].

The TREAT algorithm [28] is a modification of the Rete algorithm which does not store the results of join operations in β memories. Thus a TREAT

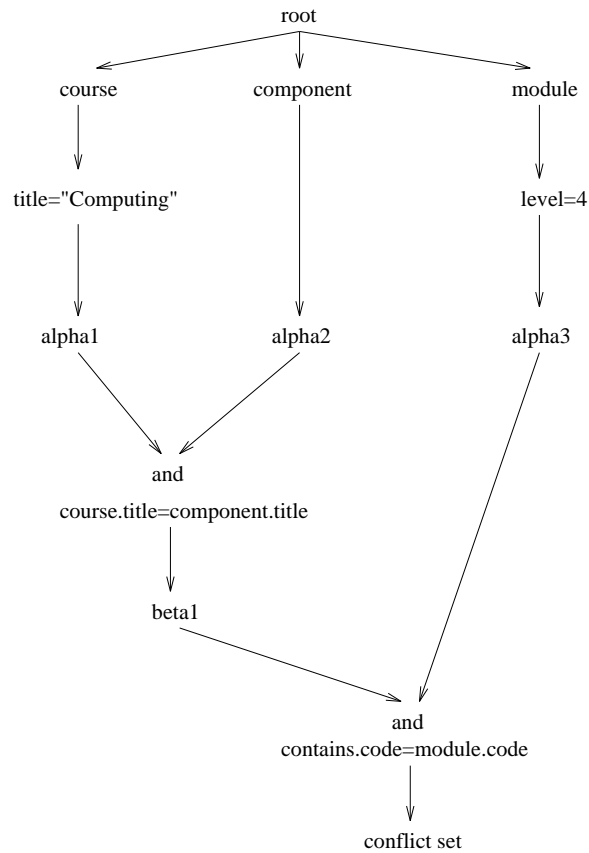


Figure 2: Rete network for example rule

network is a simplification of the Rete network which only includes α memory nodes, as shown in figure 3 for the rule for which the Rete network is given in figure 2.

The TREAT match phase operates by passing information down through the network from the root to the α nodes in the same way as Rete. The new tuple can then be used in a relational query with the α nodes of the other relations in a rule. For example, when a *course* is added to the TREAT network in figure 3, evaluation of the condition involves executing the following query, where *alpha1_add* is a relation containing the insertion(s) to *alpha1*.

```

if crs.title = cont.title and mod.code = cont.code
  from crs in alpha1_add, cont in alpha2, mod in alpha3

```

This query is much less expensive to evaluate than the query in the condition, as the relations being joined are generally much smaller than the base tables from which they are derived, although there is a need for a query optimiser to plan the most effective join order. The advantages of TREAT over Rete include reduced memory usage (no β memory support) and faster processing of tuple deletion, as there is no need to keep β memories up to date. Empirical [28, 7] and simulation-based [36] studies have shown that TREAT consistently outperforms Rete, with the added benefit of reduced space overheads.

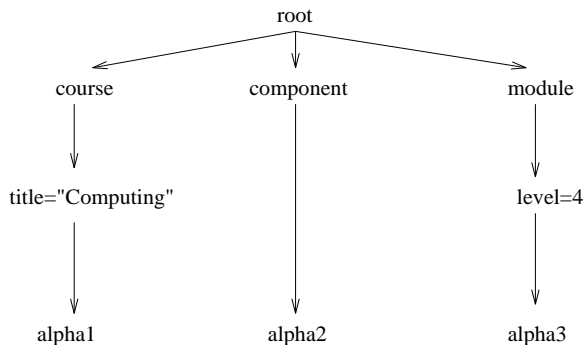


Figure 3: TREAT network for example rule

A number of recent projects have looked at tuning the TREAT algorithm to give better performance for particular rule bases. For example, in Ariel [25] work has been done on the efficient processing of single relation condition evaluation, and experiments have shown that in some cases it is appropriate to derive rather than store α memory nodes. In [18] an algorithm is presented which takes as input a set of rules and generates as output a network which takes into account space/time trade-offs for the particular rule-base and database in question.

3 Adapting Match Algorithms to Object-Oriented Models

In adapting well established production rule algorithms for relational databases for use with OODBs, it is worthwhile to consider how certain trade-offs affect performance. In particular, discrimination networks such as those described in section 2 build upon the premise that the storage of intermediate results in α or β memories leads to faster testing of rule conditions. The specific benefits derived from each of these categories of storage are:

- **α memory:** stores the result of selections on extensional relations which must subsequently be joined to new tuples inserted into the database.

This reduces the size of the relations which must be joined during the match phase, and enables single-table selections to be performed once when the tuple is inserted rather than each time a rule is processed.

- **β memory**: stores the result of joins between alpha memory nodes which in turn must subsequently be joined with other alpha memory nodes, which also reduces the size of the relations which must be joined during the match phase.

The specific costs associated with each of these categories of storage are:

- **α memory**: every distinct reference to a relation in a rule potentially leads to a new alpha memory node being created. Where there are significant numbers of rules, this additional storage requirement can be much larger than the original database (e.g. in [7] examples are given of α memory sizes in five benchmark applications, which range from approximately the size of the database to 12 times the size of the database). Deleted tuples must also be removed from α memories.
- **β memory**: every distinct join in the condition of a rule is potentially materialisable, which could lead to storage overheads vastly greater than database size, and which requires maintenance both on insertion and deletion of tuples from base tables.

Such storage requirements are unlikely to be supported in main-memory, and in many large applications this effectively precludes the use of discrimination networks.

The essence of the approach taken in Rete/TREAT is that the additional storage occupied by the α memory is justified by the gains in speed which result from joining smaller relations and performing fewer selections on retrieved tuples. In the context of OODBs, however, data is no longer arranged in tables which have to be joined, but rather as objects with direct references to other objects. Thus, given an object, finding related objects involves pointer chasing, rather than value-based matching of tabular structures. In discrimination networks, this calls into question the need for α memories which exist in large part to reduce join cardinalities. In the TREAT network in figure 3 the insertion of a *course* tuple leads to a test of its *title* attribute to see if it qualifies to become a member of *alpha1*, and if so, a join is carried out of that tuple with the base relation *component* (which is the same as *alpha2*) and with the tuples from the relation *module* which have a *level=4* (as stored in *alpha3*).

The corresponding information can be stored in an OODB using the schema shown in figure 4. In this context, there is no need for an intermediate construct (corresponding to the *component* relation), as the relationship is represented by a direct reference from one related class to the other. Using a revised OSQL notation [4], the example rule can be expressed as:

```
if crs.title = "Computing" and
```

```

mod in component(crs) and
mod.level = 4
from crs in course, mod in module
do <action>

```

In such a rule, the insertion of a *course* object plus its attributes would lead to a test on that object to see if it has a *title=Computing*, and the retrieval (by pointer chasing from the new instance) of the corresponding *mod* objects, which can then be tested for a *level=4*. Such a condition evaluation may not be made significantly more efficient by the storage of α memory data, and will sometimes be faster than the join required by TREAT (for a more detailed analysis of performance see section 4.2).

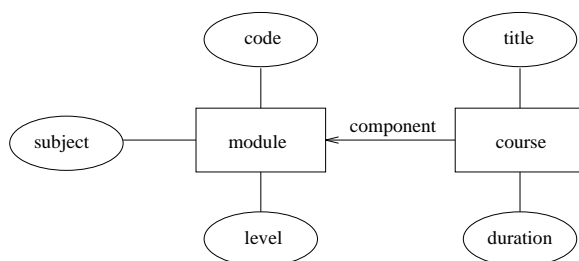


Figure 4: Class diagram for course database

A similar argument can be applied to β memories. In the Rete network of figure 2 using the same example rule, greatest benefit can be derived from the β memory when a new *module* is inserted into the database. In this case, the *module* is first tested to see if it has a *level=4*, and if so, it is joined to *beta1* to yield new data for the conflict set. In the object-oriented approach, insertion of a new *module* leads to a check that it has a *level=4*, the retrieval of the appropriate *course* objects (by pointer chasing) and a test for courses which have a *title=Computing*. In this case, the OODB needs to follow the *inverse* of the *component* relationship between *course* and *module*; the maintenance of stored inverses is necessary for good performance in this context. However, this is likely to be a modest space overhead compared with the storage of α or β memories, and the space occupied does not grow as new rules are added. Furthermore, inverses are useful for other tasks, such as query evaluation. As in the case of α memories, the processing required in the object-oriented approach will sometimes be less than is required for the join to the β memory, and again there is no storage overhead that is specific to rule processing.

The following section describes an approach to the implementation of production rules in OODBs. The following are the principal lessons which are exploited as a consequence of analysing related work on production rule algorithms for relational systems:

- The classical matching algorithms for production rules in relational databases successfully avoid recomputing the entire condition of a rule by materialising some or all of the intermediate tables which would be generated during evaluation of the rule’s condition.
- The extensive use of materialisation means that such algorithms have significant storage overheads, and that effort must be expended to update materialised values in response to insertion, deletion or modification operations on tables referenced by rules.
- The materialisation of intermediate values may be less effective or necessary in object-oriented models due to the direct storage of relationships and navigational approach to retrieval.

4 Implementing Production Rules In OODBs

This section outlines how the considerations presented in the previous section can be accounted for in an implementation of condition-action rules using an ECA-rule system. The approach presented generates a set of ECA-rules for each production rule. These ECA-rules are used to support incremental evaluation of production rule conditions by monitoring the changes which affect the truth of the condition of a production rule. It would be possible to support a similar strategy to incremental condition evaluation by changing the kernel of the database, but the implementation strategy presented here has the advantage of requiring minimal extensions to the underlying database, and as such is an addition to the range of database facilities supportable by ECA-rules, earlier examples of which are reported in [9, 10, 16, 30]. This section also includes an analysis of the performance of the approach (subsequently referred to as OPRA – Object Production Rule Algorithm), comparing it with TREAT.

4.1 Implementing Production Rules using ECA Rules

The strategy presented here is based upon a prototype implementation in the OODB ADAM [31], and utilises two existing components of ADAM – a query optimiser, which is a modified version of that presented in [33, 26], and an active rule system which supports ECA-Rules [17].

The algorithm for generating ECA rules from production rules is given in figure 5. The algorithm, given a production rule R , generates a set of ECA-rules which implement R .

The following are used in the algorithm:

- The functions *condition* and *action*, which retrieve the condition and action parts of a production rule; the function *class_of*, which, given a variable, returns the class of the variable.
- The event *new(c)*, which is raised whenever a new instance of class c is created; the event *insert(c,a)*, which is raised whenever a value is inserted

```

proc process_rule(R)
  for each variable v ∈ condition(R) do
    c := class_of(v)
    events := { new(c) }
    for each attribute a of c ∈ condition(R) do
      events := events ∪ { insert(c,a), update(c,a) }
    end-for
    for each e ∈ events do
      cond := optimise(condition(r),{v})
      act := optimise(action(r),{v} ∪ output(condition(r)))
      eca := on e if cond do act
      insert(eca)
    end-for
  end-for
run_rule(r)
end

```

Figure 5: OPRA: Object Production Rule Algorithm, generating ECA-rules.

as the attribute *a* of an instance of class *c*; and the event *update(c, a)*, which is raised whenever the attribute *a* of class *c* is modified.

- The operator \cup , which performs the union of two sets, and the operator \in which denotes set membership.
- The function *optimise* which, given a query language statement and a set of variables appearing in the statement that will be known when it is evaluated, generates an efficient evaluation strategy for the statement.
- The function *output* which, given a query, returns a set of variables which are bound by that query.
- The procedure *insert* which, given an ECA-rule, adds that rule to the rule base.
- The function *run_rule* which, given a condition-action rule, executes the action of the rule in all circumstances for which the condition is true.

In this approach, a set of ECA rules is responsible for monitoring the database and responding to changes which are relevant to the production rule. The function `run_rule` is invoked when a production rule is created to effect the initial consequences of the creation of the new production rule – the ECA rules monitor changes to the database consistent with the presence of the rule, but do not themselves cause the production rule to be activated when it is first inserted.

As an example of the ECA rules generated by the above algorithm, the example rule from section 3 is now revisited:

```

if crs.title = "Computing" and
  mod in component(crs) and
  mod.level = 4
  from crs in course, mod in module
do <action>

```

Maintenance of this rule requires monitoring changes to instances of the classes *course* and *module*, and in particular the attributes *title* and *component* of *course*, and the attribute *level* of *module*. For example, if an existing *course* object has its *title* changed to *Computing*, then the condition may become true. The ECA rule generated to monitor this possibility is:

```

on update to title of course
if crs.title = "Computing" and
  mod in component(crs) and
  mod.level = 4
  from crs in updated course, mod in module
do <action>

```

This rule is triggered by a modification to the *title* of any *course* object. The condition tests to see if the *updated* course object has *Computing* as its new *title*, and is also related to an appropriate *module* object. As indicated in section 3, this condition evaluation can be carried out by selective navigation from the updated object, using a plan generated by the optimiser. Thus not only is re-evaluation of the whole condition of the rule avoided, the selective retrieval of objects directly related to the updated object and relevant to the condition leads to competitive evaluation costs, as described in section 4.2.

It is also worthwhile to compare this approach with that of Rete/TREAT when information is deleted from the database. For example, if a *course* is deleted from a Rete/TREAT network, then every occurrence of that course in α memories or in the conflict set must be deleted. Furthermore, in Rete, it is also necessary to remove every tuple in a β memory which has been constructed using the deleted tuple. By contrast, in OPRA there is no replicated state associated with the algorithm, and thus there is no associated overhead maintaining such state on object deletion.

4.2 Performance Analysis

This section presents several formulae which have been used to estimate the performance of OPRA, and to compare it with TREAT. The simulation model for TREAT is based upon that given in [36], and the model for OPRA has been developed along similar lines. Only IO costs are considered, and the aim is to give a broad estimate of the likely performance of OPRA compared with a leading relational approach.

In TREAT, when a new tuple is inserted into the database, it is necessary to identify any changes to the conflict set which result from this insertion. In a rule which has a condition which involves joins, the new tuple must be joined

with the α memory of a related relation, and the results of the join joined with the other α memories associated with the rule. Where the α memory of a rule is stored using a B+-tree, the IO cost of the join of the intermediate relation i with the α memory relation α_k is:

$$\text{cost of index lookup on join column} + \text{cost of fetching corresponding tuples} \\ (\#i * LB(\alpha_k) + Yao(\#\alpha_k, y)) * IO$$

where $\#i$ returns the number of tuples in the relation i , LB is the number of levels in the B+-tree associated with its parameter, y is the number of tuples from α_k which will be in the result of the join, IO is the cost of an IO operation, and $Yao(x, y)$ is a function which estimates the number of blocks touched when retrieving y tuples from a relation containing x tuples [36]. The cpu cost associated with the construction of intermediate results and the comparison of key values is ignored in this model.

In OPRA, when a new attribute value is inserted, it is necessary to navigate along relationship links from the updated object for each rule which both visits the relevant object type and has a condition which explores relationships. In the simulation model presented here, it is assumed that the object identifiers (oids) used to identify objects uniquely are implemented in a way which requires an index lookup to identify the disk address of a specific object. Furthermore, to emphasise the similarity with the TREAT model, it is assumed that this $oid \rightarrow address$ mapping is implemented using a class-specific B+-tree. This is essentially how an OODB might be implemented on top of a relational storage system, as in [19].¹ In this context, the IO cost of following a relationship between the intermediate object collection i and the class C_k is:

$$\text{cost of oid lookup for each attribute} + \text{cost of fetching corresponding objects} \\ (\#i * LB(C_k) + Yao(\#C_k, y)) * IO$$

where $\#i$ returns the number of objects in the collection i , LB is the number of levels in the B+-tree associated with its parameter, y is the number of objects from C_k which are related to the objects in i by the relationship being followed, and IO is the cost of an IO operation. This approach is likely to be slower at processing retrievals than TREAT for two reasons: the index into C_k may be larger than the index into the α memory, as only a subset of the tuples in a base table are stored in an α memory; the total number of objects retrieved in OPRA will be at least as great as the number of tuples retrieved by TREAT, as every tuple in the α memory satisfies the selection condition which must be applied *after* the retrieval in OPRA.

Where oids are implemented as structured addresses (i.e. there is no level of indirection in the $oid \rightarrow address$ mapping – an overview of issues relating to object management in OODBs is given by [5]), the IO cost of following relationships in OPRA is:

¹Indeed, this particular implementation is equivalent to TREAT in which all α memories are virtual, and thus gives an indication of the expected performance if ECA-rules are used to support production rules in a relational system.

$$Yao(\#C_k, y) * IO$$

This revised formula is based on the earlier one for OPRA, but removes the additional index lookup for object dereferencing. In this case, the IO cost for TREAT will be less than for OPRA whenever the number of additional disk accesses associated with retrieval of unwanted objects in OPRA is greater than the number of disk accesses associated with index lookup in TREAT. There is also the additional cpu cost in OPRA of evaluating single-class selection conditions after object retrieval.

Table 1 outlines the IO performance, for the insertion of a single data item, of the three approaches for which models have been presented above – TREAT, OPRA with index-based oids (OPRA_I), and OPRA with reference-based oids (OPRA_R). In the simulation, which is based upon the insertion of a single data item, it is assumed that each α memory contains 33% of the tuples in the base relation, that the number of tuples (objects) in each of the relevant base relations (classes) is 1,000,000, and that each join yields 2 times as many tuples as are in the intermediate data set. Clearly TREAT is faster at processing insertions than OPRA in this case, the principal difference coming from the fact that fewer tuples are retrieved from the α memory by the join than objects retrieved by navigation, because of the selection condition applied before tuples are added to the α memory.

TABLE 1			
Joins	TREAT	OPRA _I	OPRA _R
1	0.08	0.11	0.09
2	0.16	0.27	0.20
3	0.31	0.59	0.45
4	0.63	1.22	0.93
5	1.28	2.51	1.89

Table 2 presents simulation results for different relationship cardinalities, allowing each join in TREAT to generate intermediate relations of different size. The database and rule base are the same as for Table 1, except that rules are assumed to perform 2 joins. Again, TREAT is associated with less IO activity, but the quantity for OPRA is never vastly greater than for TREAT, despite the considerably smaller storage requirements.

There are, however, a number of additional factors which favour OPRA rather than TREAT:

- The space overheads of TREAT make its use impractical in applications with large numbers of rules operating over large databases. Even where the rule base is of modest size, the space overhead associated with the α memory can be very high. For example, in [7] an example is given of a constraint propagation rule base with 35 rules, where the α memory is approximately 5 times the size of the underlying database.

TABLE 2			
Cardinality	TREAT	OPRA _I	OPRA _R
1	0.10	0.13	0.09
2	0.16	0.27	0.21
3	0.24	0.47	0.39
4	0.34	0.73	0.63
5	0.46	1.05	0.93

- When evaluating conditions, there will often be more joins required in TREAT than the maximum length of pointer chain in OPRA, as an additional relation must be created in the relational model to represent many:many relationships. Such relationships can be represented directly in OODBs using a multi-valued attribute and its inverse. For example, the rule in section 2.2 operating over the simple course database requires 2 joins in TREAT, but only one level of navigation in OPRA. The simulation model above was re-run, assuming that every second relation being joined represented a many:many relationship which was representable in the OODB by a multi-valued property and its inverse. In this case, the performance predicted for OPRA_R was almost identical to that of TREAT.
- The processing of deletes is quicker in OPRA than in TREAT, as there is no need to remove data from α memories. The IO cost of deleting a tuple from an α memory in TREAT is $LB(\alpha_k) + \mathcal{L}$. As this cost has to be paid for every α memory of which a deleted tuple is part, the cost of processing a delete in a TREAT system increases as new rules are added to the rule base.
- The reduced active data set size in OPRA is likely to lead to more effective use of cached data than in TREAT, with a corresponding reduction in IO activity. For example, in a database where α memories are on average 33% of the size of their base tables and do not overlap, there are 10 relations containing 100,000 tuples each of 100 bytes, there is a cache of 50Mb, each rule performs on average 2 joins, and the relationship cardinality is 2, the simulation yields Table 3. In this example database, the size of the α memories in TREAT reaches the size of the underlying database when only 10 rules are present.

5 Consequences and Requirements

5.1 Coupling Modes

An important design issue for ECA-rule systems is the range of coupling modes which are supported [13, 32]. The coupling mode of a rule indicates when the

TABLE 3			
#Rules	TREAT	OPRA _I	OPRA _R
05	0.11	0.13	0.10
10	0.12	0.13	0.10
15	0.13	0.13	0.10
20	0.13	0.13	0.10

condition of a rule is evaluated relative to the event, and when the action is executed relative to the condition. For example, in Starburst [37] the event-condition coupling is *deferred* and the condition-action coupling is *immediate* – this means that rule conditions are not evaluated until the end of a transaction, and that the action of a rule is executed as soon as possible after its condition has evaluated to true.

The notion of a coupling mode is not normally explicitly addressed in the literature on condition-action rules – where an implementation is supported by mapping to an ECA rule system, the standard approach would be to use *immediate* coupling for both the condition and the action. However, the motivation for coupling modes in ECA rule systems is also relevant to database production systems (e.g. it may be desirable to process rules which are maintaining integrity constraints at transaction commit time rather than earlier in order to allow temporary violation of constraints within the transaction). Future work will experiment with the range of coupling modes supported by the active rule system on which the prototype has been built [17] in the context of production rule applications.

5.2 Optimisation

Both OPRA and TREAT are highly dependent upon an optimiser planning efficient query evaluation strategies when carrying out incremental evaluations of rule conditions. A feature of Rete is that overlapping parts of rule conditions lead to overlapping discrimination networks. A similar effect can be achieved in OPRA using a multi-query optimiser on rule conditions, the effectiveness of which will be assessed in future work based upon an active extension of the deductive object-oriented database described in [3].

5.3 Language Issues

The algorithm for generating ECA rules from production rules presented in section 4 considers only productions with positive conditions. It is normal, however, also to support negative conditions in production rule languages such as OPS-5 [20] and Ariel [25]. For example, the following rule monitors `Computing` courses which contain no level 4 modules:

```
if crs.title = "Computing" and
```

```

not exists (
    mod in component(crs) and
    mod.level = 4
    from mod in module)
from crs in course
do <action>

```

This can be supported by a modest extension to OPRA, where ECA rules are generated which monitor changes to course titles, changes to module levels, changes in course components, and deletion of modules. In general, extensions to the condition expression language can be implemented using modest amounts of additional code, although certain characteristics can lead to relatively slow condition testing (e.g. independent subqueries, aggregate operations). Such constructs may benefit from selective storage of intermediate results.

5.4 Priorities

In production rule systems developed for expert system applications, it is common for the conflict resolution phase of the recognise-act cycle to select rules based upon the recency of the update which caused the rule instantiation to be added to the conflict set. This approach directs the forward chaining process in such a way that a chain of inference is followed to completion before alternatives are tried (i.e. the search tree is built depth-first rather than breadth-first).

In database systems, however, it is common for priorities to be defined at the rule level, either by specifying the relative order of rules or an absolute rule priority. Priorities can be given to rules to reflect their role, or to remove nondeterministic behaviour [1]. Where production rules are supported by a mapping onto ECA-rules, as described in section 4.1, it is straightforward to map production rule priorities onto the priority mechanism of the ECA system. It is less clear how to support a recency-based rule selection mechanism where the implementation is by mapping.

6 Conclusions

This paper has presented an approach to the implementation of production rules in OODBs. While there has been considerable attention given to support for active behaviour using ECA-rules in OODBs, there have been few proposals for the incorporation of condition-action rules into object databases. The implementation strategy presented, which involves a mapping onto ECA-rules, provides an additional use for ECA-rules in OODBs. That production rules can be implemented using an underlying ECA-rule system is not a surprising result, but the practicalities of this approach have not been investigated before. What emerges is that the task is extremely straightforward, and a prototype implementation has been developed for ADAM [31, 17] in only a few hundred lines of code. However, ease of implementation is not sufficient to justify the

approach, as concern could certainly be expressed that implementing production rules using ECA-rules is likely to yield prohibitively poor performance. To address this issue a number of performance analyses have been carried out which compare the mapping approach to a leading algorithm for implementing production rules in the relational context. What emerges from this comparison is a classic speed/space trade-off. The relational algorithm, TREAT, materialises intermediate results to reduce run-time processing, whereas the mapping approach, OPRA, navigates from updated objects. It has been shown that materialisation does generally give more efficient implementation of the match phase, but that the difference between the two approaches is often less than might have been expected, given the amount of space which is occupied in order to achieve the speed-up in TREAT. Indeed, it transpires that the space overheads for materialisation are extremely high, and that for large rule bases these are likely to preclude the use of such techniques. Thus the conclusion is that not only can production rules be supported by a mapping onto ECA-rules, but the resulting implementation offers overall performance which is competitive with the best existing algorithms.

Acknowledgements: The author would like to thank Oscar Diaz for helpful comments on an earlier draft of the paper. The prototype implementation was greatly simplified by a number of extensions to the core ADAM OODB implemented over the years by Oscar Diaz, Khoa Doan, Zhuoan Jiao and Graham Kemp. Active database research involving the author has benefited from funding from the UK EPSRC (Grant GR/H43847) and the EU Human Capital and Mobility Network ACT-NET.

References

- [1] R. Agrawal, R. Cochrane, and B. Lindsay. On maintaining priorities in a production rule language. In G. Lohman, A. Sernadas, and R. Camps, editors, *Proc. 17th VLDB*, pages 479–487. Morgan-Kaufmann, 1991.
- [2] E. Baralis and J. Widom. An algebraic approach to rule analysis in expert database systems. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th VLDB*, pages 475–486. Morgan-Kaufmann, 1994.
- [3] M.L. Barja, N.W. Paton, A.A.A. Fernandes, M.H. Williams, and A. Dinn. An Effective Deductive Object-Oriented Database Through Language Integration. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf. on Very Large Data Bases (VLDB)*, pages 463–474. Morgan-Kaufmann, 1994.
- [4] D. Beech. A foundation for evolution from relational to object databases. In J.W. Schmidt, S. Ceri, and M. Missikoff, editors, *Advances in Database Technology (EDBT)*, pages 251–270. Springer-Verlag, 1988.
- [5] E. Bertino and L. Martino, editors. *Object-Oriented Database Systems*. Addison-Wesley, 1993.
- [6] H. Branding, A. Buchmann, T. Kudrass, and J. Zimmermann. Rules in an Open System: The REACH Rule System. In N.W. Paton and M.H. Williams, editors, *Rules in Database Systems*, pages 111–126. Springer-Verlag, 1994.

- [7] D.A. Brant, T. Grose, B. Lofaso, and D.P. Miranker. Effects of database size on rule system performance. In G.M. Lohman, A. Sernadis, and R. Camps, editors, *17th VLDB*, pages 287–296. Morgan-Kaufmann, 1991.
- [8] J. Campin, N.W. Paton, and M.H. Williams. A Structured Specification of an Active Database System. *Information and Software Technology*, 37(1):47–61, 1995.
- [9] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *16th Intl. Conf. in Very Large Data Bases, Brisbane*, pages 567–577. Morgan Kaufman, 1990.
- [10] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf on Very Large Data Bases*, pages 577–589. Morgan Kaufmann, 1991.
- [11] S. Ceri and J. Widom. Managing semantic heterogeneity with production rules and persistent queries. In R. Agrawal, S. Baker, and D. Bell, editors, *19th Intl. Conf on Very Large Data Bases*, pages 108–119. Morgan Kaufmann, 1993.
- [12] S. Chakravarthy, E. Anwar, L. Maugis, and D. Mishra. Design of Sentinel: an object-oriented DBMS with event-based rules. *Information and Software Technology*, 36(9):555–568, 1994.
- [13] U. Dayal, A.P. Buchmann, and D.R. McCarthy. Rules are objects too: A knowledge model for an active object oriented database system. In K.R. Dittrich, editor, *Proc. 2nd Intl. Workshop on OODBS*, volume 334, pages 129–143. Springer-Verlag, 1988. Lecture Notes in Computer Science.
- [14] C. Delobel, M. Kifer, and Y. Masunaga. *Deductive and Object-Oriented Databases (Second International Conference DOOD'91, Munich)*. Springer-Verlag, Berlin, 1991.
- [15] O. Diaz. Deriving rules for constraint maintenance in an object-oriented database. In I. Ramos A.M. Tjoa, editor, *Proc. Intl. Conf. on Databases and Expert Systems DEXA*, pages 332–337. Springer-Verlag, 1992.
- [16] O. Diaz and N. Paton. Sharing behaviour in an object oriented database using a rule-based mechanism. In M.S. Jackson and A.E. Robinson, editors, *Aspects of Databases - Proc. British National Conference on Databases (BNCOD 9)*, pages 17–37. Butterworth-Heinemann Publishers, 1991.
- [17] O. Diaz, N. Paton, and P.M.D. Gray. Rule management in object oriented databases: a uniform approach. In G.M. Lohman, A. Sernadas, and R. Camps, editors, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 317–326. Morgan Kaufmann, 1991.
- [18] F. Fabret, M. Regnier, and E. Simon. An Adaptive Algorithm for Incremental Evaluation of Production Rules in Databases. In R. Agrawal, S. Baker, and D. Bell, editors, *19th Intl. Conf. on Very Large Data Bases*, pages 455–466. Morgan Kaufmann, 1993.
- [19] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan. Iris: An Object-Oriented Database Management System. *ACM Transactions on Office Information Systems*, 5(1):48–69, November 1987.
- [20] C.L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence*, 19:17–37, 1982.

- [21] S. Gatzui and K.R. Dittrich. Events in an active object-oriented database. In N.W. Paton and M.H. Williams, editors, *Rules in Database Systems*, pages 23–39. Springer-Verlag, 1994.
- [22] S. Gatzui, A. Geppert, and K. Dittrich. Integrating active concepts into an object-oriented database system. In P. Kanellakis and J.W. Schmidt, editors, *Proc. 3rd Workshop on Database Programming Languages*. Morgan-Kaufmann, 1991.
- [23] N.H. Gehani and H.V. Jagadish. ODE as an Active Database: Constraints and Triggers. In R. Camps G.M. Lohman, A. Sernadas, editor, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 327–336. Morgan Kaufmann, 1991.
- [24] N.H. Gehani, H.V. Jagadish, and O. Shmueli. Event specification in an active object-oriented database. *ACM SIGMOD*, pages 81–90, 1992.
- [25] E.N. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proc. SIGMOD*, pages 49–58. ACM, 1992.
- [26] Z. Jiao and P.M.D. Gray. Optimisation of Methods in a Navigational Query Language. In [14], pages 22–42, 1991.
- [27] G. Kiernan, C. de Maindreville, and E. Simon. Making Deductive Databases a Practical Technology: a step forward. In H. Garcia-Molina and H.V. Jagadish, editors, *Proc. ACM SIGMOD Conf*, pages 237–246, 1990.
- [28] D.P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems. In *Proc. AAAI*, pages 42–47, 1987.
- [29] W. Naqvi and M.T. Ibrahim. Rule and knowledge management in an active database system. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, pages 58–69. Springer-Verlag, 1994.
- [30] N. Paton, O. Diaz, and M.L. Barja. Combining active rules and metaclasses for enhanced extensibility in object-oriented systems. *Data and Knowledge Engineering*, 10:45–63, 1993.
- [31] N.W. Paton. ADAM: An Object-Oriented Database System Implemented in Prolog. In M. H. Williams, editor, *Proceedings of the Seventh British National Conference on Databases (BNCOD 7)*, pages 147–161, Edinburgh, July 1989. Cambridge University Press.
- [32] N.W. Paton, O. Diaz, M.H. Williams, J. Campin, A. Dinn, and A. Jaime. Dimensions of active behaviour. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, pages 40–57. Springer-Verlag, 1994.
- [33] N.W. Paton and P.M.D. Gray. Optimising and Executing Daplex Queries Using Prolog. *The Computer Journal*, 33(6):547–555, 1990.
- [34] T. Sellis, C-C Lin, and L. Raschild. Coupling production systems and database systems: A homogeneous approach. *Trans. on Knowledge and Data Engineering*, 5(2):240–255, 1993.
- [35] M. Stonebraker, A. Jhingran, J. Goh, and S. Potamianos. On rules, procedures, caching and views in database systems. In *Proc. ACM SIGMOD*, pages 281–290, 1990.
- [36] Y-W Wang and E.N. Hanson. A Performance Comparison of the Rete and TREAT Algorithms for Testing Database Rule Conditions. In *Proc. Data Engineering*, pages 88–97. IEEE, 1992.

- [37] J. Widom, R.J. Cochrane, and B.G. Lindsay. Implementing Set-Oriented Production Rules as an Extension to Starburst. In R. Camps G.M. Lohman, A. Ser-nadas, editor, *17th Intl. Conf. on Very Large Data Bases, Barcelona*, pages 275–286. Morgan Kaufmann (ISBN 1-55860-150-3), 1991.