[LN+93] B. Lings, A. Narayanan, H. Gan and L. Foster. Active databases and causal reasoning. *Research Report 277*, Department of Computer Science, University of Exter, 1993.

[PS+93] N. Pissinou, R. T. Snodgrass, R. Elmasri, I. S. Mumick, M. T. Özsu, B. Pernici, A. Segev, B. Theodoulidis and U. Dayal. Towards an Infrastructure for Temporal Databases: Report of an Invitational ARPA/NSF Workshop, *Technical Report TR 94-01*, University of Arizona, 1994.

[PS+94] N. Pissinou, R. T. Snodgrass, R. Elmasri, I. S. Mumick, M. T. Özsu, B. Pernici, A. Segev, B. Theodoulidis and U. Dayal. Towards an Infrastructure for Temporal Databases: Report of an Invitational ARPA/NSF Workshop, *ACM SIGMOD Record*, vol. 23(1), pages 35-51, March, 1994.

[BL94] M. Berndtsson and B. Lings. ER1C: A study in Events and Rules as 1st Class. Presented at *Dagstuhl-Seminar 9412 on Active Databases*, Germany, 1994.

[CA+94] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes and P. Jeremes. Object-Oriented Development: The FUSION Method. *Prentice Hall*, 1994.

[CB+89] S. Chakravarthy, B. Blaustein, A. Buchman, M. J. Carey, U. Dayal, D. Goldhirsch, M. Hsu, R. Juahari, M. Livny, D. McCarthy, R. McKee and A. Rosenthal. *HIPAC: A research project in active, time-constrained database management*, Final Technical report. Technical Report XAIT-89-02, Xerox Advanced Information Technology, August 1989.

[CCS94] C. Collet, T. Coupaye and T. Svensen. NAOS Efficient and modular reactive capabilities in an Object-Oriented Database System. To *appear in Proceedings of the 20th International Conference on VLDB*, Santiago, Chili, September, 1994.

[CK+93] S. Chakravarthy, V. Krishnaprasad. E. Anwar and S.-K Kim. Anatomy of a Composite Event Detector. *UF-CIS Technical Report TR-93-039*, University of Florida, 1993.

[DBM88] U. Dayal, A. Buchmann, and D. McCarthy. Rules are Objects Too: A Knowledge Model for an Active, Object-Oriented Database Management System. In *Proceedings of the 2nd International Workshop on Object-Oriented Database Systems*, Bad Muenster am Stein, Ebernburg, West Germany, Sept. 1988.

[DJ94] O. Diaz and A. Jaime. EXACT: an EXtensible approach to ACTive object-oriented databases. Presented at *Dagstuhl-Seminar 9412 on Active Databases*, Germany, 1994.

[DPG91] O. Diaz, N. Paton and P. Gray. Rule Management in Object Oriented Databases: A Uniform Approach. In *Proc. of the 17th International Conference on VLDB*, pages 317-326, Barcelona, Spain, Sept. 1991.

[Eri93] J. Eriksson. CEDE: Composite Event DEtector in an Active Object-Oriented Database, *Master's thesis*, Department of Computer Science, University of Skövde, Sweden, 1993.

[GD93] S. Gatziu and K. R. Dittrich. Events in an Active Object-Oriented Database System. In *Proceedings of the 1st Workshop on Rules in Database Systems*, Edinburg, pages 23-39, August 1993.

[GJS92] N. Gehani, H. V. Jagadish and O. Smueli. Event Specification in an Active Object-Oriented Database. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, pages 81-90, San Diego, June 1992.

[GJS93] N. Gehani, H. V. Jagadish and O. Smueli. COMPOSE: A System for Composite Event Specification and Detection. In *Advanced Database Concepts and research Issues*, edited by Nabil R. Adam and Bharat Bhargava, Lecture Notes in Computer Science, Springer Verlag,1993.

[IK93] H. Ishikawa and K. Kubota. An Active Object-Oriented Database: A multi-paradigm approach to constraint management. In *Proc. of the 19th International Conference on VLDB*, pages 467-478, Dublin, Ireland, 1993.

6 Conclusions

In this paper we have introduced a refined ECA model, referred to as E_CCA, which supports an event algebra with conditions. The need to incorporate the idea of logical events into traditional ECA-rules is a consequence of the need to specialise events in order to more closely model real-world semantics, and in particular conceptual level events. In most approaches, specialization of events has to be modelled in rule conditions. This makes the maintenance of events and rules more complex, and also leads to unnecessary triggering of rules.

By adopting the semantics of E_CCA it becomes possible to efficiently support specialization of events. This has been verified through the use of simple cost formulae, and the examination of example event structures. Importantly, the introduction of E_CCA rules is shown to give increased expressiveness and better structuring of event and rule definitions. The increased expressiveness is demonstrated with an example, exploring Sentinel's parameter contexts, which cannot be duplicated using event algebras not based on E_CCA.

The ideas expressed in this paper are current being explored through the ACOOD prototype system at the Högskolan i Skövde, Sweden. ACOOD is currently being enhanced to support the E_CCA approach. We have at this moment implemented: an efficient rule subscription mechanism, i.e. rules associated with specific events; an Event superclass, PrimitiveEvents, MethodEvents and AbstractEvents; an event subscription mechanism between AbstractEvents and PrimitiveEvents.

REFERENCES

[AMC93] E. Anwar, L. Maugis and S. Chakravarthy. Design and Implementation of Active Capability for an Object-Oriented Database. *UF-CIS Technical Report TR-93-001*, University of Florida, 1993.

[Ber93] M. Berndtsson. Reactive Object-Oriented Databases: or what causes events in ACOOD? In *Dagstuhl-Seminar-Report 62 (abstract), Seiminar on Formal Aspects of Object Base Dynamics,* page 19, Germany, April, 1993.

[Ber94] M. Berndtsson. Reactive Object-Oriented Databases and CIM. To *appear in Proceedings of the 5th International Conference on Database and Expert Systems Applications,* Athens, Greece, September, 1994.

[BL92] M. Berndtsson and B. Lings. On Developing Reactive Object-Oriented Databases. *IEEE Data Engineering, Special issue on active databases,* vol 15 (1-4), pages 31-34, December 1992.

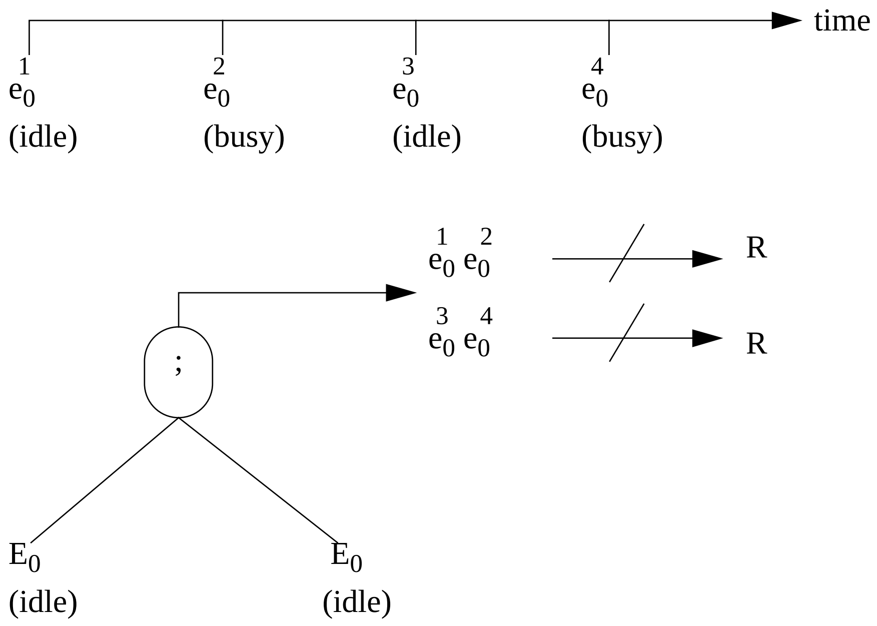

Fig. 7 Chronicle context, non-logical events
(after [CK+93])

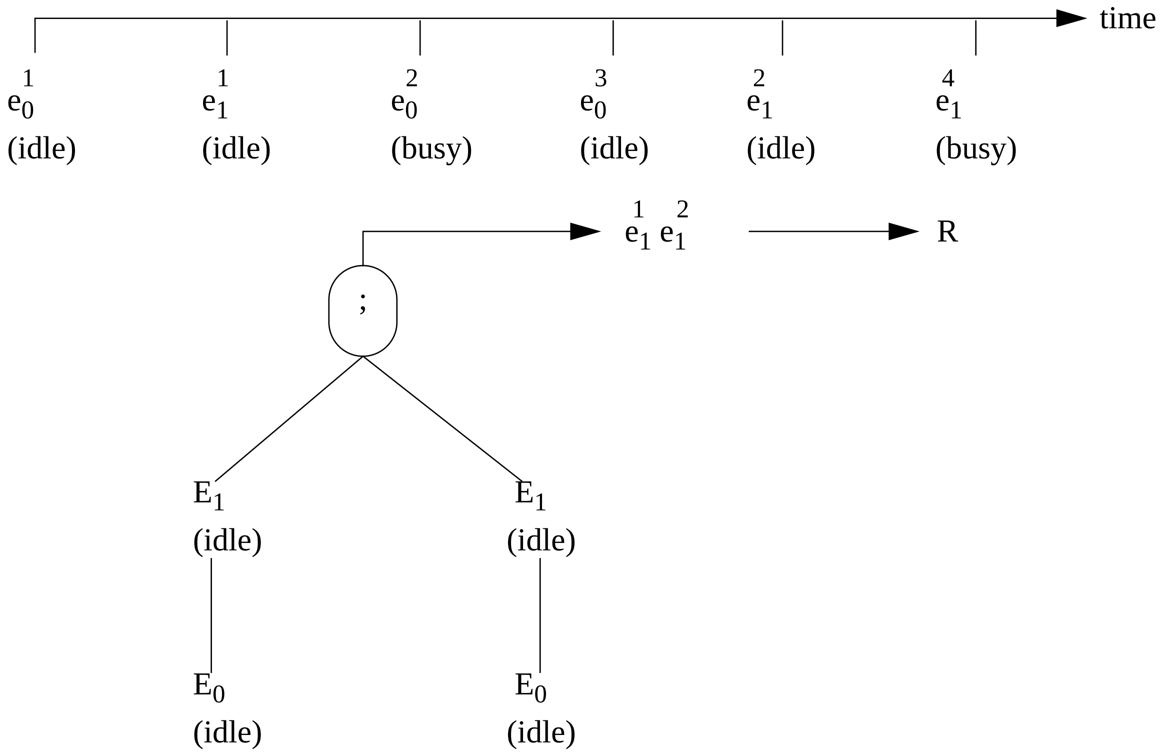

Fig. 8 Chronicle context, logical events
(after [CK+93])

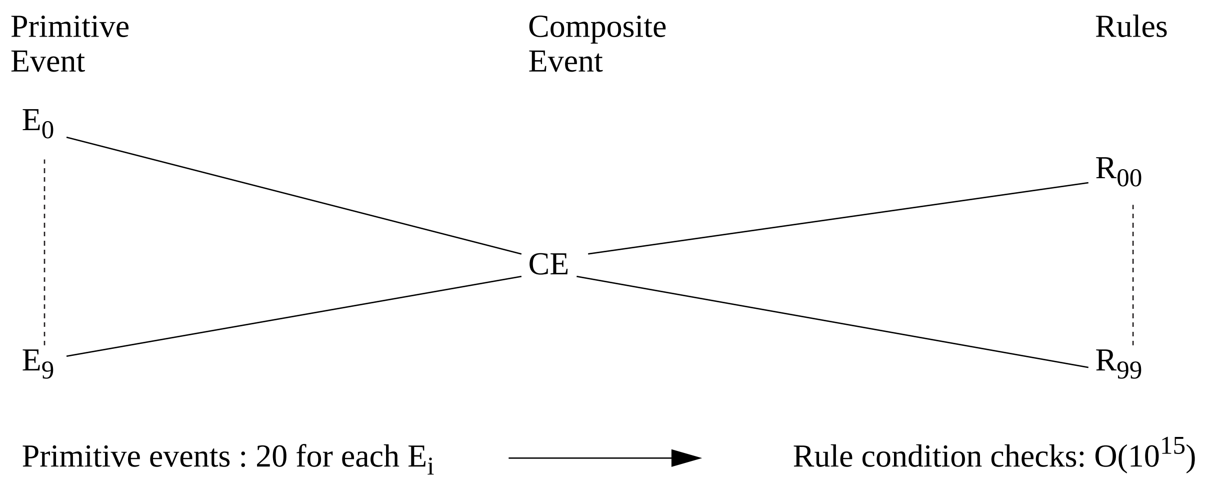
Primitive Event
Composite Event
Rules
E_0
E_9
CE
R_{00}
R_{99}
Primitive events : 20 for each E_i
Rule condition checks: $O(10^{15})$

Fig. 6 Composite events and non-logical events

Another relevant question at this stage is: what happens when the 'recent' or 'chronicle' contexts of Sentinel are used instead of 'continuous'? Rather than perform a similar analysis, we attack this issue in a different way. That is, rather than simply dwell on efficiency we reflect on the semantics of composite events. This becomes important when moving away from 'continous' context because, in 'recent' and 'chronicle' modes, not all composites are built. This means that it is too late to leave logical event detection to the rule stage, by which time a number of potential composites will have been pruned (never generated) whereas others will have been built irrespective of whether the component events are logically connected. This means that there is a significant lack of control over which event occurrences are paired with which in the various selective contexts.

Consider the simple scenario of a composite which is defined as a sequence of two primitive events. Suppose also that there is a rule associated with the composite event which will only fire if the composite is of two events matching, i.e. with the (single) *status* argument of 'break-down' in each case. In the scenario of Figure 7, no such rule firing will occur. If, however, logical events were used (Figure 8) then the alternative interpretation, resulting in a single rule firing, will occur. In this scenario, the captured semantics are of a single abstract event with an unconditional associated rule firing. Note that the default interpretation is also available, by using 'chronicle' mode with primitive, non-logical events.

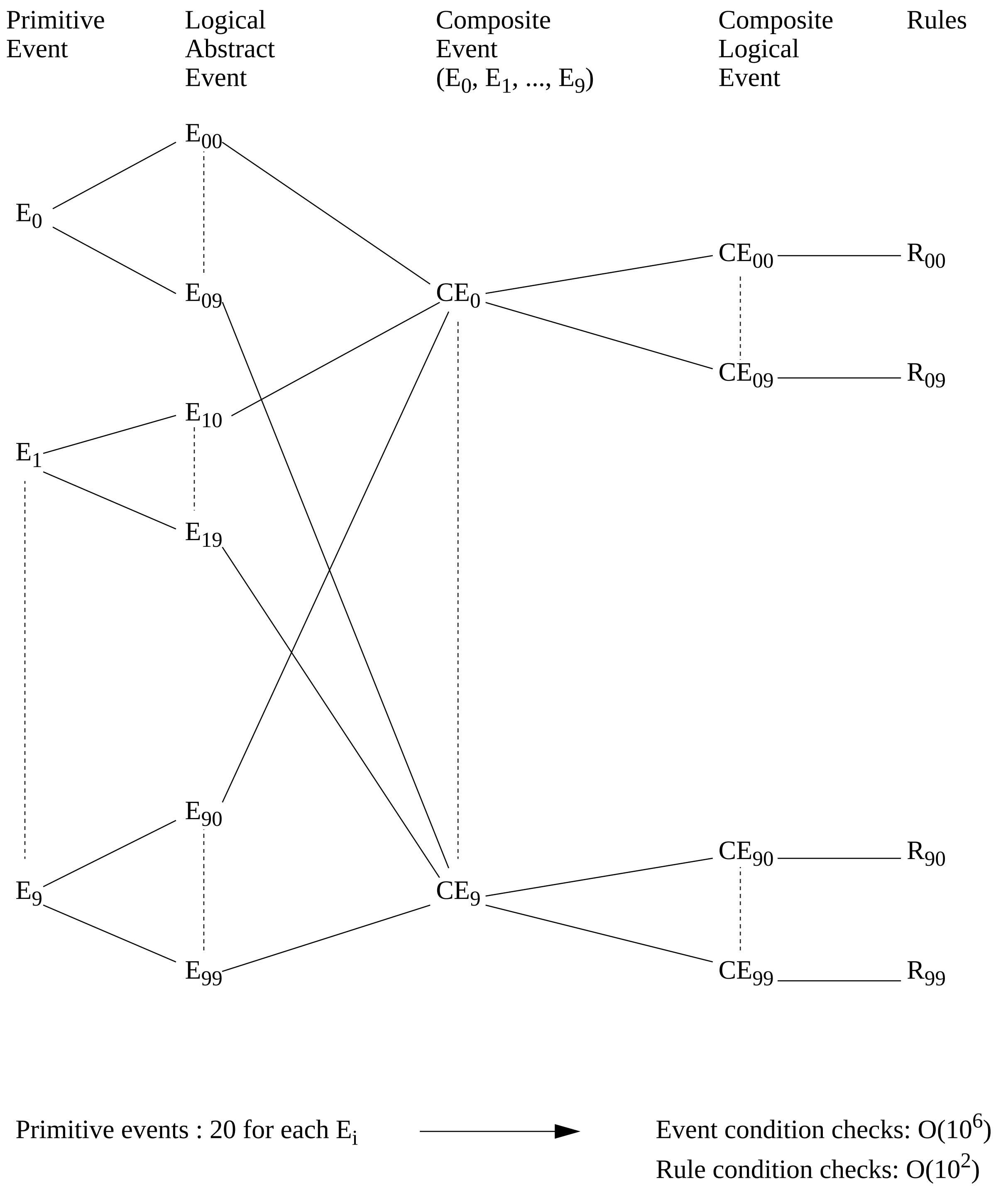

Fig. 5 Composite events and logical events

5.3 Relevant Questions

We believe that the argument concerning abstraction of events, and consequent event and rule structuring, is a powerful one for promoting E_CCA over ECA. However, we have also identified a potential efficiency argument. A relevant question at this stage is: Is it not true that much of the previous context checking has merely been moved from rule conditions to logical events? Thus, the same amount of work still has to be done by logical events, although run-time rule triggering may have been reduced.

A counter to this argument can easily be generated. For this, we assume composite logical events (a feature of the design of ACOOD which has not yet been implemented in the prototype). For simplicity, we assume ten primitive events, each with ten logical events as specializations. We further assume that there is an equal (1 in 10) probability associated with each logical event. We now hypothesise ten composite events, each a conjunction over ten logical events, one based on each primitive. We then specialise each composite into ten logical composites, and associate rules with each. This whole scenario is depicted in Figure 5.

We can analyse these event and rule sets in a number of ways. For the purposes of this exercise we assume that there are twenty occurrences of each primitive event, and that the system is using a 'continuous' context [CK+93] for composite event detection.

With logical events, we can expect (making a simplifying assumption that each primitive event raises only one logical event) that there will be two occurrences of each logical event. Under the scenario of Figure 5, this will lead to 2^{10} (approx. 10^3) occurrences of each composite event. By a similar argument, we can expect 10^2 occurrences of each logical composite event. Finally, this will lead to checking each rule set against 10^2 logical events. This is, of course, only a part of the story. Much of the condition checking will have taken place at the event level. If we take this into account, we see that there are, in fact, of the order of 10^6 event conditions to check in this scenario. Overall, then, we have a cost dominated by this factor.

It is not difficult to see that, in the case of non-logical events, we have a very different scenario with a combinatorial explosion of composite events leading to approximately 10^{15} events to check for possible rule firing (Figure 6).

ACOOD adopts rules associated with specific events. This reduces run-time rule checking further than rules indexed by classes. Thus, the values for $COST_{Ei}$ and URC_{Ei} are the same as for SAMOS.

$$COST_{Ei} = \frac{R_{E1}}{R} = \frac{5}{250} = 0,02$$

$$URC_{Ei} = 1 - \frac{R_{E1}}{R_{E1}} = 1 - \frac{5}{5} = 1 - 1 = 0$$

Both ACOOD and SAMOS are based upon rules associated with specific events, which means that only rules interested in the generated event are involved in run-time rule checking. However, since ACOOD is based on E_CCA whereas SAMOS adopts the traditional ECA model, there is a significant difference when considering the value for URC_{EiCj}.

$$URC_{EiCj} = 1 - \frac{R_{EiCj}}{R_{EiCj}} = 1 - \frac{1}{1} = 1 - 1 = 0$$

Since the E_CCA model provides us with support for both logical events and rule conditions, there is no need to perform any context check in rule conditions. For example, in SAMOS we would have to check the context in five different rule conditions in order to find R_{EiCj}. In ACOOD we avoid this rule triggering by performing the context check with logical events.

Table 2: Run-time rule checking

	$COST_{Ei}$	URC_{Ei}	URC_{EiCj}
SAMOS	0.02	0	0.8
Ode	0.1	0.8	0.96
ACOOD	0.02	0	0

DEFINE EVENT E1 **ON** E0

IF status='break-down'

etc.

DEFINE EVENT E2 **ON** E0

IF status='idle'

Then we can define the corresponding rules as

DEFINE RULE <rule_definition_oid>

ON <event>

IF <condition>

THEN <action>

DEFINE RULE R1

ON E1

IF ....

THEN NotifyProduction-Manager(Object* oid, char* status)

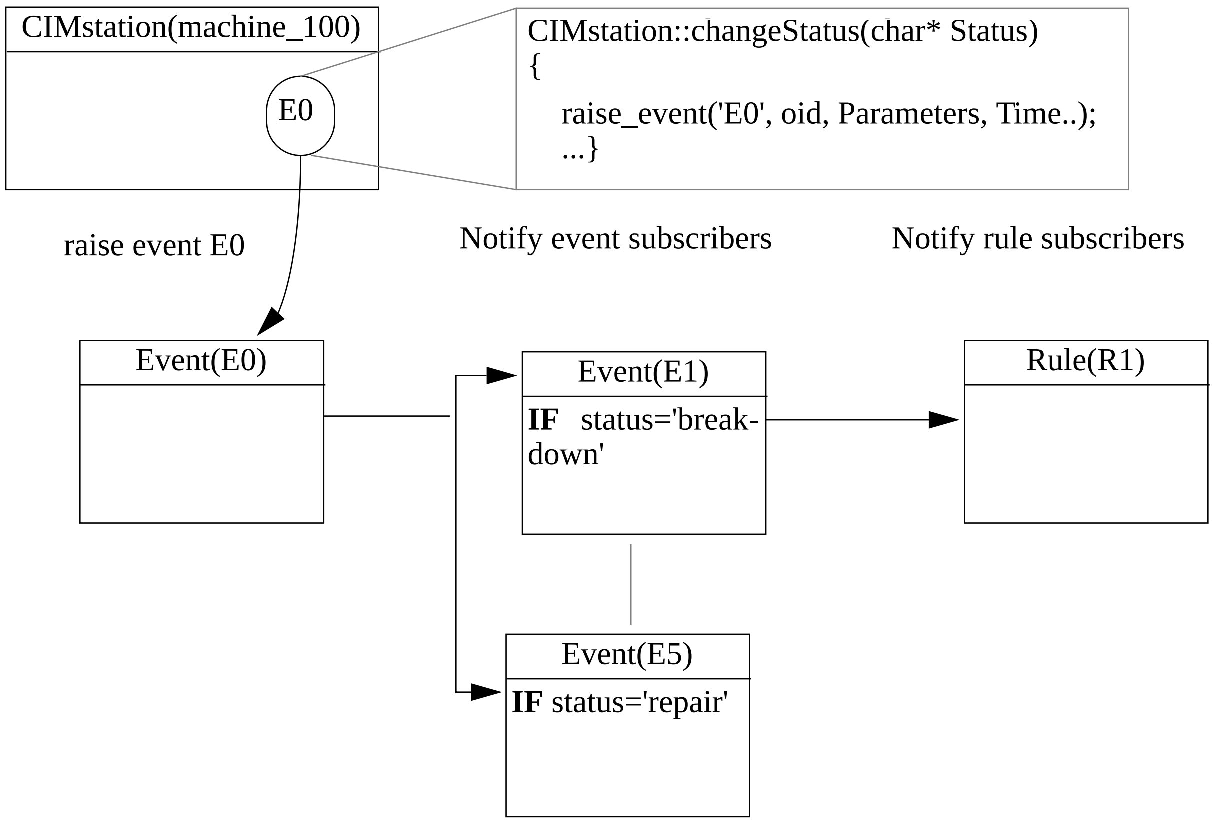
```
CIMstation::changeStatus(char* Status)
{
    raise_event('E0', oid, Parameters, Time..);
    ...}
```

Fig. 4 Example of E_CCA model

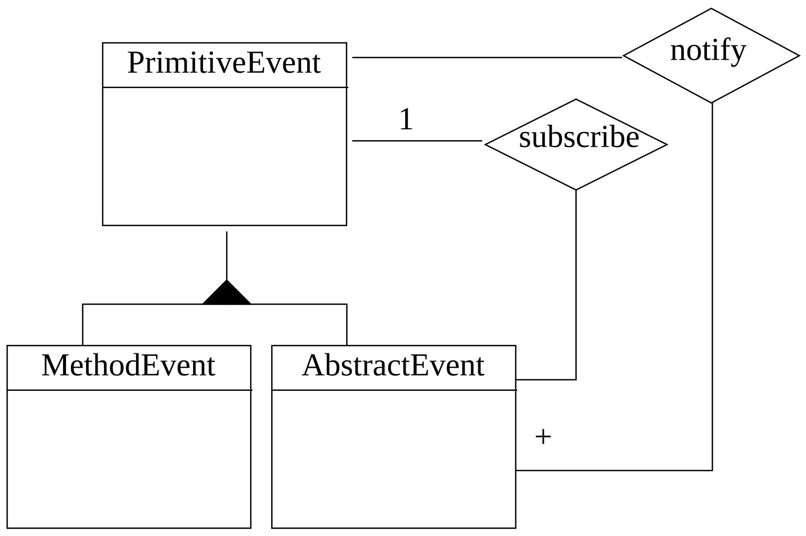

Fig. 3 Event subscription for logical abstract events

Let us apply our approach to the previous example. The whole scenario is depicted in Figure 4. First, a MethodEvent needs to be defined which is associated with the invocation of method changeStatus in class CIMstation. A MethodEvent in ACOOD is defined as

DEFINE EVENT <event_definition_oid> **ON** [**BEFORE** | **AFTER**] <class><method>

{**IF** boolean-expression}

DEFINE EVENT E0 **ON AFTER** CIMstation::changeStatus(char* status)

Given the definition of event *E0* we can define specialized versions of *E0* by introducing logical AbstractEvents and event subscription. Each specialization of *E0* requires that a corresponding logical abstract event is defined.

DEFINE EVENT <event_definition_oid> **ON** <PrimitiveEvent>

{**IF** boolean-expression}

5.1 ACOOD - an active object-oriented system based on E_CCA

ACOOD is a prototype active object-oriented system being built on top of ONTOS™, a commercial OODBMS which has C++ as its base language. Events and rules in ACOOD are represented as first class objects. Motivations for representing rules as first class objects can be found in [DBM88], [AMC93], and events as first class objects in [DPG91], [AMC93]. By representing both events and rules as first class we have a framework for supporting runtime management of rules and events. However, the active system must be very carefully designed, in order to avoid overhead. In order to reduce unnecessary rule checking we adopt the idea of rules associated with specific events [Ber94]. This idea, referred to as subscription, is also adopted in a number of other proposals, including SAMOS [GD93], EXACT [DJ94] and NAOS [CCS94].

5.2 Specializaton of events

To accommodate specialization of events in ACOOD we introduce an event subscription mechanism between abstract events and primitive events. An AbstractEvent in ACOOD inherits a logical condition from the Event superclass. This means that the event algebra is extended to allow logical conditions, which are separated from rule conditions. A primitive event in ACOOD can thus be specialized through abstract events. A logical event subscription between event E_i and event E_j can take place when (Figure 3)

1. The subscriber E_i is an AbstractEvent and E_j is a PrimitiveEvent.
2. The attributes of E_i form a subset of the attributes of E_j.

Furthermore, an AbstractEvent E_i can only subscribe to one E_j, if E_j is a PrimitiveEvent. After subscription has taken place the subscriber E_i will be notified every time E_j occurs. Note that a MethodEvent cannot act as a subscriber, since it is directly associated with a class. A MethodEvent acts as the original event generator and cannot signal an occurrence of itself on the basis of another event occurrence.

5 E_CCA - A new approach

As we have seen in previous sections, most active database systems are based on an event algebra which does not allow conditions. Thus, every check of event parameters has to be performed by rule conditions, which implies unnecessary triggering of rules. On the other hand, event algebras which allow computation of event parameters are either restricted as in SAMOS, only SAME(?ID) operator, or allow little or no support for runtime management of rules and events as in Ode [GJS93].

We propose a new approach, which tries to extend the advantages with current approaches, and at the same time avoid their disadvantages. The new approach has a particular advantage in that it gives a better mechanism for handling event abstractions, and so gives finer control of event semantics and better structure to event and rule bases.

Consider a scenario in which a primitive event is used to represent a conceptual-level abstract event. We may consider, for example, that we wish to monitor *unusual* maintenance problems. Initially we may be happy with the definition of *unusual* as simply a change of status from 'idle' to 'break-down'. A number of monitoring rules may be associated with this event. There may also be a number of composite events defined, to monitor second-order effects of *unusual* problems.

It is apparent that, in the above situation, there is a maintenance problem with respect to the event and rule sets if we wish to refine our definition of what constitutes an *unusual* event at the conceptual level. If, for example, we wish to redefine it as a change in status from either 'idle' or 'repair' to 'break-down' then, instead of a single redefinition of the event, we must identify and change each and every rule using the event, whether as a primitive or a composite. This is, of course, equally true for E_CA systems like Ode and the more common ECA systems, which do not allow logical events at all. These latter do not allow the capture of such abstract event definitions, and so once again the predicate governing the concept of *unusual* is redundantly duplicated throughout the rule base.

Of course, one could introduce rules in order to add the required structure. However, it is axiomatic to this research that the event algebra should be specialized to allow efficient and effective definition of event semantics rather than leaving the designer to use the general rule system.

Adopting the formulae in [Ber94] :

$$COST_{E1} = \frac{R_{Cm}}{R} = \frac{25}{250} = 0,1$$

$$URC_{E12} = 1 - \frac{R_{E12}}{R_{Cm}} = 1 - \frac{5}{25} = 1 - 0,2 = 0,8$$

Since triggers (rules) in Ode are defined in Class definitions, we need to notify every trigger which is defined within class CIMstation about the event occurrence. This means that 10 percent of the rules are involved when event *E1* is generated. Furthermore, 80 percent of the triggers in class CIMstation receive a notification about event *E1* in which they are not interested. The major disadvantage with this approach is that the implementation is somewhat *ad hoc* [AMC93] and provides little support for runtime management of rules and events. Neither rules nor events are represented as first class objects in Ode [GJS92], [GJS93].

$$URC_{EiCj} = 1 - \frac{R_{EiCj}}{R_{Cm}} = 1 - \frac{1}{25} = 1 - 0,04 = 0,96$$

As much as 96 percent of the search performed for triggers which are specifically interested when event *E1* reports a machine break-down, i.e. R_{EiCj}, is unnecessary.

Table 1: Run-time rule checking

	$COST_{Ei}$	URC_{Ei}	URC_{EiCj}
SAMOS	0.02	0	0.8
Ode	0.1	0.8	0.96

To summarize, efficient run-time rule (trigger) checking is most of all dependent upon the underlying architecture for rule subscription. For example, adopting rules indexed by classes implies that more rules are involved in run-time rule checking than in rules associated with specific events. Thus, we contend that logical events can become much more efficient when they are combined with a more efficient rule subscription approach such as rules associated with specific events.

4.2 Logical Events

The event specification language in Ode [GJS92] is based on event-action rules where the condition is folded into the event part. Composite events and basic events (primitive) in Ode can include an optional predicate that is used to *hide* some occurrences. This provides the user with a slightly different set of tools to define rules and events. Other approaches for active databases do not explicitly support an event algebra which can contain conditions. A composite event may well indicate that events E1 and E2 are constituent parts, but there is likely to be some condition on the validity of associations between instances. For example, one may insist that a composite event has not occurred unless both events relate to the same object; or occur in the same transaction; or relate to two objects sharing a common property. SAMOS [GD93] recognises the necessity for the first two of these conditions by introducing Same(?ID); in Ode it is extended to full condition evaluation.

The scenario which was introduced in the previous section can be modelled in Ode as five triggers with logical events:

```
after changeStatus(char* status) && status='break-down'
    ==> NotifyProductionManager(Object* oid, char* status)
after changeStatus(char* status) && status='idle' ==> ...
after changeStatus(char* status) && status='busy' ==> ...
after changeStatus(char* status) && status='unloading' ==> ...
after changeStatus(char* status) && status='repair' ==> ...
```

Rules (triggers, constraints) in Ode are defined in class definitions. This means that their rule subscription is different from the approach taken in SAMOS. Ode indexes rules by classes in order to achieve an association between rules and events.

are performed, i.e. $URC_{E1} = 0$. Triggering a rule means that the rule is subject to condition evaluation. Thus all five rule sets will be evaluated in order to detect if the context, i.e. value of the status parameter, satisfies the condition part. If the parameter of method changeStatus is equal to break-down, then we have a situation in which we have notified: one rule that is interested in the generated event *E1* and matches the context (break-down); four rules which are interested in the generated event *E1*, but do not match the context (e.g. idle, busy) in which the event was generated.

By extending the previously used formulae we can describe this by:

$$URC_{EiCj} = 1 - \frac{R_{EiCj}}{R_{Ei}} = 1 - \frac{1}{5} = 1 - 0,2 = 0,8$$

URC_{EiCj} = Unnecessary rule checks which are performed when event E_i is generated in context C_j.

As much as 80 percent of the performed rule checking is unnecessary, since the contexts (event parameters) are checked in rule conditions. However, using the ECA model gives the user little choice in reducing rule checking further, since the user has no other choice than to perform the context check in rule conditions. The above scenario is not unique to SAMOS, it is inherited by all approaches which are based upon the traditional ECA model, including Sentinel [AMC93], ADAM [DPG91], HiPAC [CB+89], EXACT [DJ94], and NAOS [CCS94].

The scenario is made even worse if composite events are also under consideration. It may well be that the system also wishes to signal an event of a machine break-down which has been followed by a missed-deadline associated with that machine; or repeated break-downs of a machine in a given period. These require logically connected event components, where the logical connection may be arbitrarily defined (rather than simply based on common OID, TransactionID etc.). Many 'non-events' may be generated if no such filter (or defining condition) is applied, and a significant increase in rule processing would follow. Event specification languages which do not allow conditions cannot avoid the problem of rules performing the context check. Composite events can be used to specify the situation, but they cannot perform any context check unless the event algebra allows conditions which are separated from rule conditions.

is specifically interested when event *E1* indicates a machine break-down (R_{EiCj}).

In forthcoming sections we will refer to the example above and the following actual parameters:

$R = 250$, $R_{Cm} = 25$, $R_{Ei} = 5$, $R_{EiCj} = 1$.

For the purpose of this paper we can define one example rule in SAMOS as follows[1]:

DEFINE EVENT E1 **AFTER**.CIMstation.changeStatus(char* status)

DEFINE RULE R1

ON E1 **IF** status='break-down' **AND** ....

DO NotifyProductionManager(Object* oid, char* status)

Rule subscription in SAMOS is achieved by associating rules with specific events. This means that, when an occurrence of event *E1* is generated, all rules that have subscribed to event definition *E1* will be triggered, i.e. *R1*, *R2*, .., *R5*. Adopting the formulae in [Ber94] we can describe this as:

$$COST_{E1} = \frac{R_{E1}}{R} = \frac{5}{250} = 0,02$$

$$URC_{E1} = 1 - \frac{R_{E1}}{R_{E1}} = 1 - \frac{5}{5} = 1 - 1 = 0$$

where

$COST_{Ei}$: The share of rules which are involved in rule checking when event E_i is generated

URC_{Ei} : Unnecessary rule checks which are performed when event E_i is generated

As we can see, only those rules which are interested when event E_i is generated are involved in run-time rule checking, i.e. 2 percent. This is also reflected in that no unnecessary rule checks

1. This rule can of course be defined in any rule definition language which supports ECA rules.

4 Specialization of Events

Composite events are used to model complex situations which cannot be expressed by a single primitive event. Thus, a composite event can be viewed as an abstraction of other events. There is also a concept of specialization of events. Currently, specialization of events for active databases is usually modelled in rule conditions; see, for example, Sentinel ([AMC93], [CK+93]), EXACT [DJ94], SAMOS [GD93]. For each specialization of an event, a corresponding rule condition needs to be modelled. When an event is raised all rules whose condition contains a specialization of the raised event are notified. These rules are triggered, for condition evaluation, in order to determine if specialized versions of the raised event have occurred.

4.1 ECA-Rules

The dynamic behaviour of a rule system can be captured, for cost analysis, through the following parameters [Ber94]:

R : Total number of rules.

R_{Cm} : Set of rules which are interested in when a reactive class m generates primitive events.

R_{Ei} : Set of rules which are interested in when event E_i is generated.

R_{EiCj} : Set of rules which are interested in when event E_i is generated in context C_j.

Assume that the total number of ECA-rules for a CIM application is 250 (R). The triggering event for those rules can be either a primitive event or a composite event. Among these rules we have 25 rules (R_{Cm}) which are interested in events generated from class CIMstation. Furthermore, the invocation of method changeStatus in class CIMstation is associated with event *E1*. Event *E1* is generated after the execution of method changeStatus. The parameter of method changeStatus indicates the current status mode for a CIMstation. Assume that a CIMstation can be in one of five different status modes such as break-down, idle, busy, unloading and repair. A CIMstation can change its current status by invoking the method changeStatus in class CIMstation. Each of the five different status modes may require a special response, each captured by a rule set (R_{Ei}). Thus, we introduce five rule sets, (*R1, R2, .., R5)* that can react in response to the changes in different status modes of a CIMstation. Among these five rules we can define one example ECA rule which

3 Event subscription

A composite event is a set of primitive events or composite events related by defined operators. Composite events are specified in an event language which is based on an event algebra. Recent advances in event languages for specifying composite events can be found in Sentinel [CK+93], Ode [GJS93] and SAMOS [GD93].

To establish an association between a composite event and its components, several approaches use some form of event subscription. The purpose of event subscription is similar to rule subscription. Both subscription techniques are based on the concept of subscription and notification. A rule can subscribe to an event and then receive notification when the event is raised. Similarly, a composite event can subscribe to a set of primitive events and/or other composite events, and then receive notification upon an event occurrence (Figure 2). Event subscription for composite events is provided in Jasmine/A [IK93] by an attribute in the primitive event object. The attribute *COMPOSITE_EVENT* denotes those composite events whose components include the primitive event. SAMOS [GD93] provide a similar attribute *petri_net_place* in the primitive_event_pattern object to connect primitive events and composite events. The event object in EXACT [DJ94] provides an attribute *participates_in*, which denotes the object identifiers of those composite events in which the event participates.

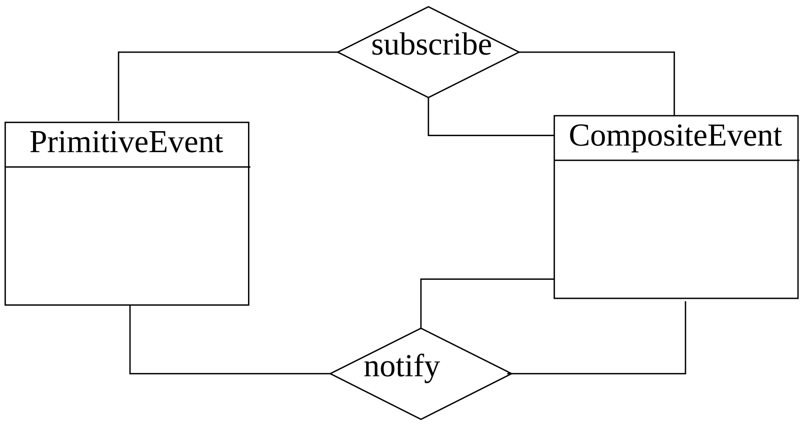

Fig. 2 Event subscription for composite events

2 Rule subscription

Rules and events in ACOOD are represented as first class objects, which means that they are subject to normal database operations like any other system defined object. Rule subscription is introduced to establish an association between rules and events. Current approaches can broadly be classified into: i) centralized rule checking, ii) rules indexed by classes and iii) rules associated with specific events.

Rules associated with specific events were introduced in [Ber94] (Figure 1). In this system, a rule can subscribe to an event object and upon an event occurrence on the object will receive a notification. The major advantage with rules associated with specific events is that only those rules which have subscribed to the generated event are notified.

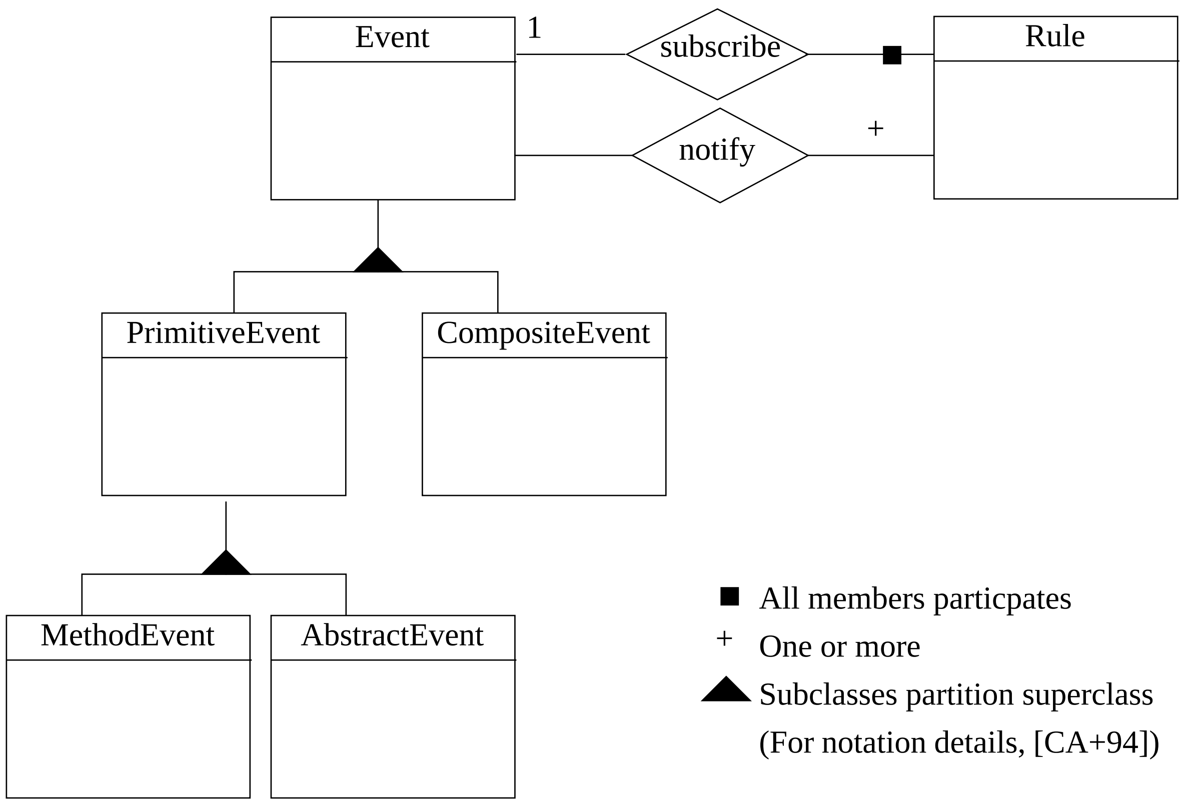

Fig. 1 Rule subscription

ment parameters are time stamp, transaction id and user id. As an example of parameters which depend on the type of event, method events have in addition the identification of the object (oid) and the parameters of the invoked method.

Ode [GJS92] introduces logical events, where the condition part of the ECA model is folded into the event part. A logical event in Ode is a basic (primitive) event extended with a predicate that is used to *mask* the occurrence of an event. The predicate can be over the event parameters and the state of any objects in the database system. One can thus state that rules in Ode are more accurately modelled as Event-Action rules rather than Event-Condition-Action rules.

The need for modelling ECA rules with logical events was first addressed in [Ber93], [LN+93] and later addressed in [PS+93], [PS+94]. In [Ber93], [LN+93] we proposed that in order to avoid situations where rules are notified about events that have not occurred, the condition for the ECA rule has to be separated into: One part which gives the necessary and sufficient condition for the event to be generated (Event_Condition); another part which adds further qualification to higher-level rule execution (Rule_Condition). A similar approach is proposed in [PS+93], [PS+94] where the event algebra can contain conditions, which are separated from the condition part in a rule. To summarize, the refined ECA model can now be described as

Event (E): is a primitive (basic) or composite event.
Event Condition (EC): is a boolean expression on the event attributes.
Condition (C): is either a boolean expression or a SQL query on the database.
Action (A): is either a database operation or an arbitrary application program that is executed.

For the purpose of this paper we will refer to the refined ECA model as E_CCA, where E_C indicates a logical event. The rest of the paper is organized as follows. Section 2 describes the rule subscription technique in our prototype. In section 3 we present how event subscription can be used to build composite events. In section 4 we discuss how specialization of events is modelled in current approaches. Section 5 introduces E_CCA rules, and explores their impact on event and rule structuring; the cost of event detection; and control over event and rule semantics. This section also briefly describes our prototype system, ACOOD. Finally, conclusions are presented in section 6.

Logical Events and ECA Rules

Mikael Berndtsson
Department of Computer Science
University of Skövde, Sweden
spiff@ida.his.se

Brian Lings
Department of Computer Science
University of Exeter, UK
brian@dcs.exeter.ac.uk

ABSTRACT

This paper presents an approach to support event-condition-action rules and logical events in an object-oriented environment. Previous approaches in active object-oriented databases support either traditional event-condition-action rules or logical events. We see the need to integrate these two concepts in order to efficiently support specialization of events.

1 Introduction

The ECA model introduced in HiPAC [CB+89] is widely used to support rules in database systems. The semantics of ECA rules are; when the event occurs, evaluate the condition, and if the condition is satisfied, execute the action. A general definition of the ECA model can be found in [PS+93], where

Event (E): is a primitive (basic) or composite event.
Condition (C): is either a boolean expression or a SQL query on the database.
Action (A): is either a database operation or an arbitrary application program that is executed.

In [PS+93] it is identified that events can have attributes, i.e. event parameters. Event parameters are usually passed on to the condition and/or action part of a rule. A separation of event attributes and event parameters is made in [GJS93]. Event attributes are considered to be immediate and referring only to the current primitive event, whereas event parameters are event attributes which have been saved. In this paper we will no use such a distinction. There is no real consensus in the active object-oriented database community on which set of event attributes is minimal. In Sentinel [CK+93] the minimal set consists solely of the identification of the object (oid) for which a primitive event is applicable. Additional event attributes for supporting method events in Sentinel are: Class, Method, Actual_Parameters and Time_Stamp. SAMOS [GD93] distinguishes between environment parameters and parameters which depend on the type of an event. Environ-

UNIVERSITY OF SKÖVDE
Department of Computer Science

Logical Events and ECA Rules

Mikael Berndtsson (spiff@ida.his.se)
Brian Lings (brian@dcs.exeter.ac.uk)

Technical Report

No: HS-IDA-TR-95-004

Contact Author
Mikael Berndtsson
University of Skövde,
Department of Computer Science
Box 408, 541 28 Skövde, SWEDEN
Tel: +46-500-464722; Fax: +46-500-464725