Formal Specification Of Active Database Functionality: A Survey

Norman W. Paton[1], Jack Campin[2], Alvaro A.A. Fernandes[2] and M. Howard Williams[2]

[1] Department of Computer Science
University of Manchester, Oxford Road, Manchester
[2] Department of Computing and Electrical Engineering
Heriot-Watt University, Riccarton, Edinburgh

Abstract. This paper reviews research on the formal specification of active behaviour, indicating both what has been done in this area, and how. The scope of different approaches is compared within a common framework, which reveals that although many aspects of active behaviour have been described formally, no single proposal covers all phenomena associated with active database systems.

1 Introduction

Database research has often been characterised by a close association between theory and practice, with formal results being used to guide and underpin the development of novel database systems. The presence of a widely accepted formal description of a language or model can encourage a more focused development effort than generally emerges from empirical work on constructs or systems.

In the area of active databases, most research has been empirical in nature, with no widely accepted formal model acting as a starting point for the development of implementations for different data models or systems. As a result, the most prominent proposals for active database systems, while sharing a range of basic notions and constructs, support widely differing languages and execution models. A preliminary, informal framework for the comparison of active database systems and applications, designed to highlight common ground and key differences, is given in [15]. However, no formal model has yet been developed that encompasses the range of facilities supported by different active database systems. Despite this, an increasing number of researchers have been working on the formal specification of different aspects of active database functionality, with a range of different aims in mind. This paper reviews a number of approaches to the formal specification of active behaviour, with a view to indicating what features have been formally specified, illustrating the formal methods that have been used, and allowing a comparison of the principal results to date. The paper is structured as follows: section 2 indicates what might be sought from a formal specification of active behaviour, sections 3 to 10 describe individual approaches from the literature, section 11 provides some pointers to other representative examples in the literature, section 12 summarises the approaches considered in this paper, and section 13 presents some conclusions.

2 Context: Active Database Systems

Active database systems are able to respond automatically to situations that arise inside or outside the database itself. The active behaviour of a database is generally described using rules, which most commonly have three components, an *event*, a *condition* and an *action*. A rule with such components is known as an event-condition-action rule, or *ECA-rule*. Such a rule lies dormant until an occurrence of the event that it is monitoring, when the rule is said to be triggered. The condition of a triggered rule is subsequently evaluated, and if true, then the rule is added to the *conflict set*, which is the store of triggered rules accessed by the scheduler which selects rule actions for execution.

The structural and behavioural characteristics of active database systems can be classified according to a number of dimensions, as outlined informally in [15]. For the purposes of this paper we classify the aspects to be described into three areas: *knowledge model*, *execution model*, and *management model*.

The knowledge model represents the syntactic view of active rules as seen by the rule programmer. This has three main facets: *Event language:* a notation in which the situations that trigger a rule can be specified; *Condition language:* a notation used to express additional constraints which must be satisfied by the database before the triggered rule is added to the conflict set; *Action language:* a notation used to specify the effect that the rule must have when it is executed.

The execution model describes how rules interact in the context of the whole database system. It has the following aspects: *Transition granularity:* the nature of the binding between event occurrences and rule activations – it is possible that an individual event occurrence will trigger a rule, or that a collection of occurrences of an event will together trigger a rule; *Coupling mode:* the temporal and causal relationship between triggering and execution – for example, it is possible that the action of a rule is executed as soon as possible after the evaluation of the condition of the rule (immediate), or at a later point, such as at end of transaction (deferred); *Priority scheme:* an ordering specifying which rules are considered first when several have been triggered at the same time (i.e. when the conflict set contains more than one rule); *Nett-effect policy:* which may allow intermediate state changes internal to a transaction to be ignored by the rule system.

The management model comprises properties of the rule base taken as a whole, as well as operations for modifying the rules: *Operations on rules:* some active databases allow rules to be created or removed, and activated or deactivated, during execution; *Termination:* whether or not the execution of a given rule set must reach a final state; *Confluence:* whether the (in general nondeterministic) rule set defines a unique final state given any context of initiation; *Equivalence Model:* a notion of semantic equivalence between rule sets, which is a prerequisite for optimisation.

Specific proposals for formal descriptions of active behaviour will be examined using the above dimensions to highlight the scope of different proposals; a comparison of the different proposals within this framework is given in section 12. Furthermore, for each approach presented the following issues are addressed:

Motivation: why the formal specification was developed.
Formalism: what formal technique was used.
Limitations: what issues were not considered, and why.

There will also be a description of each approach to give a flavour of how the formal specification has been achieved.

3 Starburst in Denotational Semantics

Motivation: to establish how readily the execution model of an existing active database system can be specified formally.
Formalism: denotational semantics.
Limitations: the specification focuses upon the execution model, and abstracts over the condition/action languages of Starburst.

The denotational model of the Starburst active rule system in [21] provides a semantics for a rule base considered as a function mapping user inputs (requests made during a transaction) and prior database states into subsequent database states. This approach emulates semantic models for conventional programming languages.

The core idea in the specification is the notion of a *set of changes* – this models both a user transaction and the effect of a rule. A rule is considered (simplifying slightly) to be a function mapping sets of changes and database states to new sets of changes and new database states. The top level of the specification is a *meaning function* which gives a denotation for a rule system (set of rules ordered by priority). Several supplementary functions are used as intermediates in the definition of this meaning function. These intermediate functions play a purely technical role; they do not correspond to data structures or natural units of behaviour in the Starburst system.

An example of this style is provided by the denotational definition of rule priority. In Starburst, this is an arbitrary partial ordering. The specification defines this informally, and makes use of it by a function *Eligible* which selects candidate rules for firing among those with highest priority:

$$
\begin{aligned}
&\mathit{Eligible} : \mathcal{RC} \times \mathcal{O} \rightarrow \mathbb{P}\,\mathcal{R} \\
&\mathit{Eligible} = \lambda\{\langle r_1, \delta_1\rangle, \ldots\langle r_n, \delta_n\rangle\}, o \bullet \\
&\qquad\qquad \{r_i \mid 1 \leq i \leq n \wedge r_i(\delta_i, \varnothing) \downarrow 1 = \mathit{true} \wedge \\
&\qquad\qquad\quad \{r_j \mid 1 \leq j \leq n \wedge r_j(\delta_j, \varnothing) \downarrow 1 = \mathit{true} \wedge \\
&\qquad\qquad\qquad r_j > r_i \in o\} = \varnothing\}
\end{aligned}
$$

$\mathcal{O}$ is the domain of rule orderings; $\mathcal{RC}$ is the domain of *sets of rule-change pairs*, pairings of a rule and an elementary change to the database. $\mathcal{R}$ is the domain of rules. The functions r_i representing the denotations of rules return a triple of values, the first of which (accessed by the $\downarrow$ selection operator) is a Boolean value indicating whether the rule has been triggered.

This notation is extremely concise; a syntactically sugared variant, like those of functional programming languages, might make for more maintainable specifications. Formal reasoning about this model is not straightforward; there are few tools available for analyzing general denotational specifications, and the general problem is intractable.

4 An Object-Oriented Framework for Specifying Active Rule Systems

Motivation: the development of a framework for formally specifying the semantics of different rule systems, with a view to allowing a detailed comparison of proposals. To date, the framework has been used to define the semantics of the Starburst[22], POSTGRES[18] and Ariel [8] rule systems [4, 5].

Formalism: Object-Z [17], an object-oriented extension of Z.

Limitations: As in the Starburst specification of section 3, details of event, condition and action languages are not included in the specifications.

The formalism of Object-Z, sharing with Z its characteristics of being a first-order theory of an evolving state, frequently gains in simplicity of expression over the functional, domain-based denotational approach, but at the expense of considerably greater length. The constraints on allowable operations on the database are introduced by *subclassing*; a very abstract description of a highly generic active database is progressively refined to converge on the functionality of specific rule systems. This incremental refinement of specifications contrasts with the denotational model, where the domains and functions are introduced all at once, and the type system of the semantics prevents extensive generalisation.

To illustrate the approach, figure 1 gives part of the Object-Z class *SRule*, a subclass of the generic class *Rule* used to model Starburst specific behaviour. The rule has stored properties a, d and r which log the nett-effect of changes to the table monitored by the rule. It is these sets of changes which, respectively, describe the append, delete or replace operations which are relevant to the event that is being monitored by the rule. The *Fire* operation tests the *Condition* of the rule, and if it is true, executes its *Action* (both *Condition* and *Action* are inherited from *Rule*); after the *Fire* operation has been carried out, the logs of changes being monitored by the rule are cleared.

The benefits of the Object-Z framework include the clarity and tailorability of the resulting specifications, which make explicit the differences between proposals for active rule systems. However, in describing the active capabilities of a range of systems, the current framework abstracts over many other features of these systems, such as the data models, query languages, etc.

5 Heraclitus: An execution model description language

Motivation: to provide a language which can be used to support alternative execution models for active database systems.

SRule

Rule

$store : SStore$

$a : SetLog[Object]$
$d : SetLog[Object]$
$r : SetLog[Object \times State]$

INIT

$emptylog$

ClearRuleLog

$\Delta(a, d, r)$

$a'.INIT \wedge d'.INIT \wedge r'.INIT$

Fire

$\Delta(adb.stores)$

$Condition \Rightarrow Action$
$\neg\ Condition \Rightarrow adb.stores' = adb.stores$
$ClearRulelog$

Fig. 1. Class *SRule* in Object-Z.

Formalism: a hybrid operational/algebraic approach.
Limitations: does not directly lend itself to formal reasoning.

Hull and Jacobs [10] describe a database programming language, Heraclitus, that is used to model active database constructs. The essential new concept in Heraclitus is the *delta*: a denotable value in the language representing a proposed change to the database. These can be inspected by the rule system and algebraically combined to return new deltas; their times of creation and manipulation are decoupled from the time when they are applied to the database (they need not be applied at all). This supports hypothetical reasoning about the state of the database; it also makes it possible to implement a variety of coupling modes with little difficulty, although this is not done in the paper.

This methodology goes some way towards implementing a desideratum identified earlier in our discussion of the denotational semantics of Starburst; providing a 'syntactic sugar' for the semantics. However, the Heraclitus language is imperative; this makes it a good prototyping language, but a declarative language with similar primitives would lend itself better to transformational reasoning.

As an example, in [10] it is shown how the execution model of Starburst can be supported using Heraclitus. The specification involves writing rules as Heraclitus functions, which are then processed by a Heraclitus program which schedules rules for execution in response to relevant updates. In this context, rules directly access the *delta*s which represent the changes to which they must respond. The following rule condition, expressed in (slightly modified) Heraclitus notation, returns *true* if there are tuples being inserted into the `tube` relation (`<+tube(tid,_,_,type)> in change`), the `type` attribute of which is neither stored in the database nor scheduled for insertion:

```
function condition@1(change,curr:delta):bool
return exists tid,type such that
              (<+tube(tid,_,_,type)> in change and
                 not tube_type(type,_,_) when curr)
```

In the above function, `change` represents the updates monitored by, but not yet processed by, the rule, and `curr` represents the updates of the current transaction. In the above rule, the `in` operation tests for the presence of an inserted tuple in the delta `change`, while the `when` operation tests to see if a relevant `tube_type` tuple will be stored in the database once the updates in `curr` have been applied.

The principal strength of the work on Heraclitus is that the formal definition of the algebra of deltas can be exploited within an implemented database programming language, thereby allowing experimentation with different flavours of active rule system.

6 A logic framework using the Event Calculus

Motivation: to establish what characteristics of active functionality can be captured within a framework based upon first order logic [6].
Formalism: event calculus [13].
Limitations: resorts to an operational semantics to describe execution model features.

This approach relies on the database case of the Kowalski-Sergot event calculus (EC) [13], which is based on a history of events. The set of all logical consequences derived from the history by the EC gives rise to a sequence of fact-sets, each of which can be viewed as the extensional part (EDB) of a deductive database (DDB). Adding an intensional part (IDB) in the form of a deductive rule (DR) set that operates over each (and all) of the clause sets characterises a complete DDB framework that only differs from the standard case in having access to the multiple states generated by the succession of events.

In the context of this paper, the main contribution of [6] is to define specification languages for events, conditions and actions in such a way that event detection, condition verification and action execution have a logical semantics.

The event specification language is a Datalog-equivalent language over the event occurrences recorded in the history. Event composition is modelled by the intensional definition of event-occurrences using DRs over the history. Ascertaining that an event (possibly a composite one) has occurred is equivalent to evaluating a query over the DDB composed of the history (as the EDB) and the DRs over it (as the IDB). This DDB defines, by logical consequence, all the events of interest in the application. Analogously, the condition specification language is a Datalog-equivalent language over the logical consequences of the DDB comprising, as its EDB, the logical consequences of the application of the EC to the history, and, as its IDB, application-specific DRs. Ascertaining that a condition holds in a database state is equivalent to evaluating a query over the induced sequence of DDBs. Finally, the action-specification language can be reduced to the monotonic appending of new event occurrences to the history.

For example, assuming `append` as a primitive action on the history, the following ECA rule (of the form $E : C \rightarrow A$) specifies that if a salary alteration occurred that raised the salary of some employee to a higher level than that of his or her manager, an action should be taken to raise the salary of the manager to the same level as that of the employee:

```
   happened(_EventId, put_salary(EmployeeSalary)@Employee),
 : holdsAt(property(Employee,manager,Manager), Now),
   holdsAt(property(Manager,salary,ManagerSalary), Now),
   ManagerSalary < EmployeeSalary, current_time(Now)
-> append(put_salary(EmployeeSalary)@Manager)
```

Note that the predicate `happened` is used to query the event history, and `holdsAt` is used to query the database. Variable bindings are assumed to flow by unification between the different parts of the rule.

The distinguishing aspect of this proposal is that it tightly integrates deductive and active rules without attempting to merge their semantics into a whole since there are grounds to believe that by merging them the whole could turn out to be less than the sum of the parts.

7 An Operational Semantics For Rule Base Analysis

Motivation: to develop a formalism for rule execution which allows reasoning about the properties of rule bases [24].
Formalism: operational semantics.
Limitations: focuses upon a single condition-action rule language.

The techniques of [24] give an operational semantics for rule systems expressed using a simple condition-action rule language. While the language is quite restrictive, user transactions can be arbitrarily complex. Only one coupling mode (immediate execution) is supported.

To give a flavour of the specification, the following function defines the effect of executing the rule α on the database state s as a result of the update u:

$$f_\alpha(s,u) = u \cup \bigcup_{(c_1 \cdots c_n) \in \varepsilon[C_\alpha(x_1 \cdots x_n)]su} U_\alpha[c_1 \cdots c_n]$$

The function indicates that the updates resulting from u and the firing of the rule α are u unioned with the updates which results from executing the action of the rule ($U_\alpha[c_1 \cdots c_n]$) for each of the tuples for which the condition of the rule is true ($\varepsilon[C_\alpha(x_1 \cdots x_n)]su$).

The execution sequences modelled by this formalism are a subset of those described in sections 3 and 4, as a constraint requires updates to accumulate monotonically. Together with syntactic constraints on rule bases which maintain this monotonic behaviour, this gives sufficient conditions to guarantee termination and confluence; however, these constraints (on variable sharing, and a prohibition on the generation of new identifiers) are too strong to be realistic for most database applications. For example, it would not be possible to write a rule with an action that increases the salary of each employee by a fixed percentage. Approaches to the analysis of rule bases are considered further in sections 8 and 9. Further work on rule analysis based upon operational semantics is described in [20].

8 Reachability analysis by graph theory

Motivation: to develop a framework within which analysis of rule bases can be conducted [1].
Formalism: graphs which represent rule execution.
Limitations: depends upon user input or unspecified analyses of rule conditions and actions.

This work, although described in the context of the Starburst rule system, presents a generic framework for establishing three properties of rule bases, namely termination, confluence and observable determinism. The analysis method proceeds in a similar way to that in section 7, but allows for more fine-grained information to be considered (e.g. by examining the effect of rules on individual tuples, and by incorporating a priority scheme to make a larger class of rule bases provably terminating and confluent). The conditions that guarantee confluence and termination (commutativity of rules) are less restrictive, but correspondingly harder to check; the authors suggest that they might be worked out interactively with user input, although the later work described in section 9 can do this automatically.

The analysis is a hybrid of syntactic and semantic reasoning; its correctness is defined in terms of *execution graphs*. The nodes n of an execution graph represent a database state D_n and a set of triggered rules TR_n, such that each rule is associated with the updates that have caused the rule to be triggered. Any edge between nodes i and j has the following characteristics [1]:

- A label r, such that $r \in TR_i$.
- There is some (possibly empty) set of operations O performed by r such that the triggered rules in TR_j can be derived from the triggered rules in TR_i by:
  1. Removing rule r.
  2. Removing some subset of the rules that can be untriggered by the action of r.

3. Adding all rules in the rule base that may be triggered by that action of r.

Properties of rule bases are then defined in terms of such graphs. For example, a rule base is confluent if every execution graph for the rule base has at most one final state.

A useful contribution of this paper is the notion of *partial confluence*, where confluence is only required to hold for part of the database state. Verification of this proceeds in the same way as for full confluence, but considers only the subset of the rules affecting the relevant part of the database.

9 Applying Relational Algebra to Rule Analysis and Optimisation

Motivation: to support analysis [3] and optimisation [2] of condition-action rules.
Formalism: relational algebra.
Limitations: execution model issues and events are not considered.

An important question in rule analysis, which must be addressed, for example, in the context of the graph analyses of [1], is 'which rule actions may activate other rules?'. A conservative approach to this problem might work on the basis that any rule action that writes to a relation can potentially activate any rule with a condition that monitors the relation. However, such an approach may detect a potential activation dependency between rules which in fact does not exist (for example, because the tuples updated by one rule always yield the value false when the condition of the potentially dependent rule is evaluated).

The work described in [3] shows how the action of one rule can be 'pushed through' the condition of another. This process of pushing an action through a condition is achieved by applying propagation rules. For example, if the rule condition contains the join $E_1 \bowtie E_2$, and the action to be propagated is represented by the insertion E_{ins} to the relation E_1, then the effect of the insertion on the result of the join is $E_{ins} \bowtie E_2$. The successive application of such propagation rules yields a relational algebra expression which may or may not be satisfiable, thereby indicating whether the action may or may not trigger the rule. In general, the satisfiability of relational expressions is undecidable, so automatic analyses must use conservative techniques which are not described in [3].

Further exploitation of relational algebra transformations for active databases is presented in [2], with a view to supporting the optimisation of rule processing. In this case, rule conditions are rewritten to exploit delta relations which represent the changes made to the database since the rule was last evaluated, exploiting knowledge as to the truth (or otherwise) of the condition of the rule when it was last evaluated. This enables a form of preprocessing to be performed on active rule conditions expressed using relational algebra, prior to the application of standard query optimisation techniques.

A strength of this work is that it effectively adapts a proven database formalism for use in active databases. The principal limitations are the close association with the relational model, and limited consideration of the impact of different execution models on rule analysis and optimisation.

10 Event Description Using Petri Nets

Motivation: to provide a formal, abstract and readily implementable description of a rich event description language [7].
Formalism: coloured Petri nets.
Limitations: no specification of features of active behaviour other than events.

The SAMOS active database has a comprehensive language for defining composite and temporal events. This comprises a wide range of primitive events (time events, transaction boundary events, events raised by method calls, user-defined events raised by explicit exception statements) and composite events built from these by composition operators (conjunction, disjunction, sequencing, negation, and counting the number of times an event has occurred).

The formalism used to model composite event detection is Petri nets; this is also used as an implementation technique. Each operator in the event algebra can be modelled as a generic fragment of net; these fragments can be glued together (making the appropriate identifications) to generate a composite net for arbitrarily complex compositions. This is represented in a manner familiar from Petri net theory – a matrix represents the static topology of the net while a vector of places represents its (dynamically changing) marking.

The following example shows how sequential composition of two events (**E1;E2**) is supported. Each occurrence of events **E1** and **E2** generates a new token. The place **H** and transition *t3* have the effect of ignoring occurrences of **E2** occurring before **E1**. This diagram can be glued into further constructs; **E1** and **E2** can be simple events or the output places of nets representing composites.

Petri nets permit optimisation by algebraic techniques; these techniques are not applied in SAMOS – instead the nets are directly executed from the original matrix/vector representation.

11 Further Work on Formal Approaches to Active Databases

Space limitations preclude detailed descriptions of other formal work on active databases. There have, however, been several significant projects not described in previous sections, a number of which are outlined below. These projects either cover ground which is beyond the scope of this paper (e.g. design), or represent alternative approaches to those presented earlier in the paper.

Harrison and Dietrich [9] provide a Datalog-based formalism which extends previous event algebras in permitting *recursive* complex events – those affecting recursively defined relations in the intensional database. Events and conditions

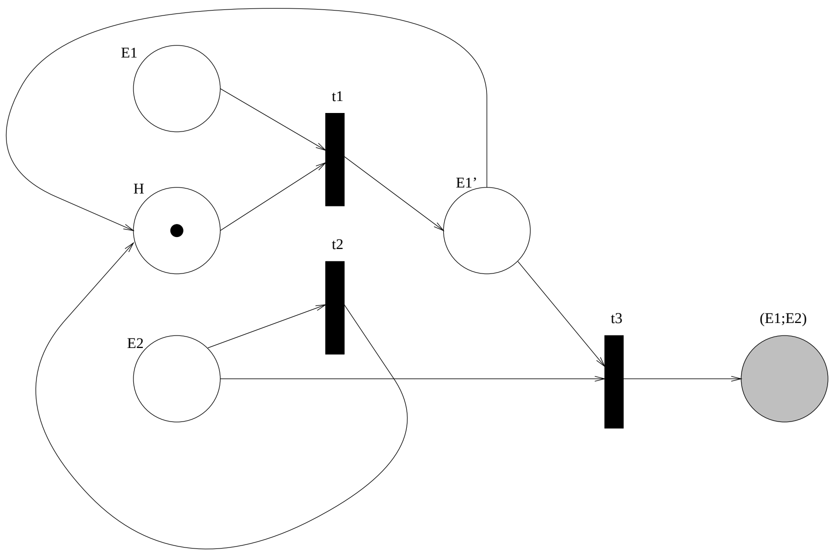

Fig. 2. Petri net for sequential composition.

are treated uniformly, with conditions permitted to examine the previous state of the database. The techniques of the paper also go beyond [7] in providing an optimisation technique for complex event detection, and handle modifications to deductive rules uniformly with modifications to data. This paper does not consider the action part of rules at all; hence issues like termination are beyond its scope.

Zaniolo [23] describes a fixpoint interpretation extending the classical operational semantics of deductive databases to provide them with a common view with active databases. The thrust of [23] is that there is a basic computational engine that can process pure deductive rules and yet simulate the effect of active behaviour. Unfortunately, it seems clear that some of the expressive power of active database systems is not easily captured in a purely declarative framework (e.g., the possibility of specifying particular coupling modes).

Teisseure, Poncelet and Cichetti [19] extend the formally defined semantic data model IFO to support mechanisms for characterising events within an application. An algorithm is presented for generating a set of ECA rules equivalent to a schema of events. This provides a structured methodology for creating rule bases, with potential to reduce design errors. However, the rule generation algorithm neither guarantees nor checks for good behavioural properties (termination and confluence), and expresses only a subset of the functionality of recent active database systems.

Raschid [16] has focused upon a rule execution mechanism for an extended relational database system with condition-action rules, and has shown how rules can be partitioned into non-interacting cliques. Correct rule systems are defined semantically as those that terminate and produce a least fixpoint as a result; a syntactic criterion for this is that they should be *partitioned* so as to limit undesirable interactions between *cliques*, which are defined by reachability. The clique construct is also used to identify nondeterminism (non-confluence).

Karadimce and Urban [12] present an approach to rule analysis based upon conditional term rewrite systems. It is shown how active rules can be mapped onto conditional term rewrite rules (CTRRs) for both execution and analysis. Execution is performed by rewriting the rules using the term rewriting system. Analysis is performed by applying results from the theory of conditional term rewrite systems to the CTRR representation of the active database rules. In this context, termination can be guaranteed when all rules have a specific syntactic property, and confluence is ensured by unifying and comparing pairs of rules in which a single term may be rewritten in different ways.

12 Comparison

The following tables summarise the strengths and weaknesses of the approaches presented in the preceding sections. A – sign indicates that an active database aspect is not represented at all; a + means that it is, but not in a manner lending itself to tractable analysis; a ++ means that formal reasoning about the aspect is supported (formal reasoning about the equivalence model amounts to optimisation capability). Entries in brackets mean that the methodology could be straightforwardly extended to describe (+) or reason about (++) the relevant feature. A less focused consideration of the merits of different formal techniques for use with active systems is given in [4].

12.1 Knowledge Model

Formalism	Event Language	Condition/Action Language
Denotational[21]	–	–
Object-Z[4]	–	–
Algebraic Semantics[11]	+	+
Event Calculus[6]	++	++
Operational Semantics[24]	+	++
Graph Theory[1]	–	–
Relational Algebra[21]	–	++
Petri Nets [7]	++	–

In certain cases, it would be possible to extend the scope of the existing specifications to include comprehensive descriptions of the event and condition/action languages (e.g. using Denotational Semantics or Object-Z), but to do so would be a significant task. In Heraclitus, reasoning about the condition/action language is complicated by the fact that algebraic and imperative constructs are used together in the complete system.

12.2 Execution Model

Formalism	Transition Granularity	Coupling Mode	Priority Scheme	Nett-Effect Policy
Denotational[21]	+	+	+	+
Object-Z [4]	+	+	+	+
Algebraic Semantics[11]	+	+	+	++
Event Calculus[6]	+	+	+	+
Operational Semantics[24]	+	+	- (+)	- (+)
Relational Algebra[21]	-	-	-	-
Graph Theory[1]	−	−	+	−
Petri Nets[7]	−	−	−	−

In a number of the above cases, only one transition granularity or coupling mode is supported, but this could often be extended to cope with description of more complex systems. In [6], the execution model has been described formally using an operational semantics, rather than within the event calculus itself. It is not clear how generally useful an ability to reason about the execution model is in isolation, but it is evident that certain aspects of the execution model do affect analysis results and optimisation potential, so access to a formal description of the execution model can be seen as important to mainstream tasks relating to active rule systems.

12.3 Management

Formalism	Operations on rules	Termin- -ation	Conflu- -ence	Equivalence Model
Denotational[21]	− (+)	−	−	−
Object-Z [4]	+	−	−	−
Algebraic Semantics[11]	− (+)	−	−	−
Event Calculus[6]	− (+)	− (++)	− (++)	− (++)
Operational Semantics[24]	− (+)	++	++	++
Graph Theory[1]	− (+)	++	++	−
Relational Algebra[21]	−	++	++	++
Petri Nets[7]	−	−	−	(++)

A number of remarks which were made in section 12.1 are also of relevance here, as, in general, reasoning about features such as termination and confluence is dependent upon event and condition/action languages being amenable to automatic analysis. In the deductive context of [6], termination and confluence analyses could exploit related work on deductive databases [14].

13 Conclusions

This paper has described and compared a range of approaches which have been adopted to the formal specification of different aspects of active behaviour. A number of points can be made in concluding:

1. Individual proposals rarely support the formal specification of a complete system, which is probably because different formalisms seem to be best suited to different tasks.
2. Despite the above point, it is relatively unusual for different formalisms to be used together (although exceptions are presented in sections 5 and 6). In other domains, it is common for multiple formalisms to be used together (e.g. for specifying the syntax and semantics of programming languages).
3. Implementations are rarely derived from formal specifications (exceptions include Heraclitus and SAMOS), although formal specifications are sometimes derived for existing systems (e.g. Starburst).
4. The potential for reasoning about specifications has rarely been exploited in implementations, and there is little evidence that working optimisers or analysers have been developed from formal specifications described in the literature.

It can be hoped that future work will: lead to formal specifications of all aspects of implemented active databases; exploit formal techniques during the implementation of systems; and develop useful tools which assist in the design and implementation of applications which use active database facilities.

Acknowledgements We are pleased to acknowledge the support of the UK EPSRC (grant GR/H43847) and the EU HC&M (ACT-NET) programme in funding active database research at Heriot-Watt. This work has also benefited from useful discussions with Andrew Dinn and Oscar Diaz.

References

1. A. Aiken, J. Widom, and J.M. Hellerstein. Behaviour of database production rules: Termination, confluence, and observable determinism. In *ACM SIGMOD*, volume 21, pages 59–68, 1992.
2. E. Baralis and J. Widom. Using delta relations to optimize condition evaluation in active databases. Technical Report Stan-CS-93-1495, Department of Computer Science, Stanford University, 1993.
3. 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.
4. 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.
5. J. Campin, N.W. Paton, and M.H. Williams. Specifying Active Database Systems in an Object-Oriented Framework. *submitted for publication*, 1995.
6. Alvaro A.A. Fernandes, M. Howard Williams, and Norman W. Paton. A Logic-Based Integration of Active and Deductive Databases, 1994. Submitted for publication.
7. S. Gatziu 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.
8. E.N. Hanson. Rule Condition Testing and Action Execution in Ariel. In *Proc. SIGMOD*, pages 49–58. ACM, 1992.

9. J.V. Harrison and S.W. Dietrich. Integrating active and deductive rules. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, pages 288–305. Springer-Verlag, 1994.
10. R. Hull and D. Jacobs. Language constructs for programming active databases. In R. Camps G.M. Lohman, A. Sernadas, editor, *Intl. Conf on Very Large Data Bases*, volume 17, pages 455–467. Morgan Kaugmann, 1991.
11. D. Jacobs and R. Hull. Database Programming With Delayed Updates. In P. Kanellakis and J. Schmidt, editors, *Third International Workshop on Database Programming Languages.* Morgan-Kaufmann, 1991.
12. A.P. Karadimce and S.D. Urban. Conditional Term Rewriting as a Basis for Analysis of Active Database Rules. In J. Widom and S. Chakravarthy, editors, *Proc. IEEE RIDE-ADS Workshop on Active Database Systems*, pages 156–162. IEEE Press, 1994.
13. R. Kowalski. Database updates in the event calculus. *Journal of Logic Programming*, 12:121–146, 1992.
14. A.Y. Levy and Y. Sagiv. Queries independent of updates. In R. Agrawal, S. Baker, and D. Bell, editors, *Proc. 19th VLDB*, pages 171–181. Morgan-Kaufmann, 1993.
15. 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.
16. L. Raschid. A semantics for a class of stratified production system programs. *J. Logic Programming*, 21(1):31–57, 1994.
17. G.A. Rose. Object-Z. In S. Stepney, R. Barden, and David Cooper, editors, *Object-Orientation in Z*, pages 59–77. Springer-Verlag, 1992.
18. 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.
19. M. Teisseire, P. Poncelet, and R. Cichetti. Towards event-driven modelling for database design. In J. Bocca, M. Jarke, and C. Zaniolo, editors, *Proc. 20th Int. Conf on VLDB*, pages 1–12. Morgan-Kaufmann, 1994.
20. L. van der Voort and A. Siebes. Enforcing confluence of rule execution. In N.W. Paton and M.H. Williams, editors, *Proc. 1st Int. Workshop on Rules In Database Systems*, pages 194–207. Springer-Verlag, 1994.
21. J. Widom. A Denotational Senmantics for the Starburst Production Rule Language. *ACM SIGMOD Record*, 21(3):4–9, 1992.
22. J. Widom and S.J. Finkelstein. Set-Oriented Production Rules in Relational Database Systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 259–270, 1990.
23. C. Zaniolo. A unified semantics for active and deductive databases. In N.W. Paton and M.H. Williams, editors, *Rules in Database Systems.* Springer-Verlag, 1994.
24. Y. Zhou and M. Hsu. A theory for rule triggering systems. In F. Bancilhon and et al., editors, *Proc. Extending Database Technology (EDBT)*, pages 407–421. Springer-Verlag, 1990.

This article was processed using the LaTeX macro package with LLNCS style